IEICE TRANS. FUNDAMENTALS, VOL.E82–A, NO.9 SEPTEMBER 1999

PAPER *Special Section on Nonlinear Theory and Its Applications*

FPGA-Based Hash Circuit Synthesis with Evolutionary Algorithms

Ernesto DAMIANI[†], **Valentino LIBERALI**[††], *and* **Andrea G. B. TETTAMANZI**[†], ***Nonmembers***

SUMMARY An evolutionary algorithm is used to evolve a digital circuit which computes a simple hash function mapping a 16-bit address space into an 8-bit one. The target technology is FPGA, where the search space of the algorithm is made of the combinational functions computed by cells and of the interconnections among cells. The evolutionary technique has been applied to five different interconnection topologies, specified by *neighbourhood graphs*. This circuit is readily applicable to the design of *set-associative* cache memories. Possible use of the evolutionary approach presented in the paper for *on-line* tuning of the function during cache operation is also discussed.

key words: *evolvable hardware, evolutionary algorithms, nonlinear circuit synthesis, FPGA*

1. Introduction

Evolutionary algorithms (EAs) are a broad class of stochastic optimization algorithms, inspired by Biology and in particular by those biological processes that allow populations of organisms to adapt to their surrounding environment: genetic inheritance and survival of the fittest. These concepts were introduced in the XIXth century by Charles Darwin [1] and are still today widely acknowledged as valid, even though complemented with further details [2].

The first proposals in that direction date back to the mid Sixties, when John Holland, of the University of Michigan, introduced genetic algorithms (GAs) [3], Lawrence Fogel and his colleagues, of the University of California in San Diego, started their experiments on evolutionary programming [4] and Ingo Rechenberg, of the Technical University of Berlin, independently began to work on evolution strategies [5]. Their pioneering work eventually gave rise to a broad class of optimization methods particularly well suited for hard problems where little is known about the underlying search space. The last development of this research thread is so-called genetic programming, introduced by John Koza, of the Stanford University [6] at the beginning of the Nineties.

Recent texts of reference and synthesis in the field of evolutionary algorithms are [7], [8].

An evolutionary algorithm maintains a population of candidate solutions for the problem at hand, and makes it evolve by iteratively applying a (usually quite small) set of stochastic operators, known as *mutation*, *recombination* and *selection*.

Mutation randomly perturbs a candidate solution; recombination decomposes two distinct solutions and then randomly mixes their parts to form novel solutions; selection replicates the most successful solutions found in a population at a rate proportional to their relative quality.

The initial population may be either a random sample of the solution space or may be seeded with solutions found by simple local search procedures, if these are available.

The resulting process tends to find globally optimal solutions to the problem much in the same way as in Nature populations of organisms tend to adapt to their surrounding environment.

After some successful applications of evolutionary algorithms to physical circuit design (partitioning, placement and routing) [9], now the same techniques are being considered also for structural design. This has given rise to an innovative field of research, called *evolvable hardware* [10]. Recently, evolutionary design automation has been applied to digital electronic design [11] and the first industrial applications have started to appear [12]. It has been proposed for analog circuits as well, with encouraging results [13].

The main motivation for the work presented in this paper is the understanding of how an evolutionary algorithm could be applied to the synthesis in hardware of a digital non-linear function. In this paper, circuits are evolved to compute a simple hashing function mapping a 16-bit address space into an 8-bit index space as uniformly as possible.

Based on the hardware architecture described in Sect. 3, an evolutionary algorithm (described in Sect. 4) is devised to evolve the hashing function (Sect. 2). Section 5 presents experimental results, while Sect. 7 envisages an application of the circuit to the design of *set-associative* cache memories.

2. Problem Statement

Integrated digital electronics is traditionally considered the ideal testbed for automated design techniques. Indeed, though to this date there is no known general technique for automatically designing VLSI digi-

Manuscript received December 10, 1998.
Manuscript revised March 23, 1999.

[†]The authors are with Polo Didattico e di Ricerca di Crema, University of Milan, Italy.
[††]The author is with the Department of Electronics, University of Pavia, Italy.

tal circuits satisfying given specifications, considerable progress has been made to automate the design of some classes of digital systems.

Over the last decades, the number of devices integrated into a single chip increased exponentially, according to Moore's law. Such a scenario leads to a demand for new design solutions, capable of managing increasing circuit complexity. A great deal of effort has been devoted by the research community to the development of suitable design automation tools.

The example presented in this paper is a hashing circuit that maps a 16-bit address space into an 8-bit index space as uniformly as possible. Circuits are evolved to compute a simple hashing function, i.e. a many-to-one mapping of a key set into an address set, having the design property of uniform filling of the co-domain. A hashing function can be used, for example, in set-associative cache memories, and in hardware cryptographic systems.

3. Hardware Environment and Behavioural Description

The hashing function is implemented using an FPGA. Regularity and modularity of FPGAs, as well as their easy-to-use development tools, make them very attractive to implement evolvable digital circuits [11].

The Xilinx SRAM-based reconfigurable FPGAs have been chosen as the reference architecture for this application [14].

This choice requires that the evolved circuits comply with the constraints imposed by the target FPGA.

At this stage of our investigation, evolution is completely carried out in software on a host computer. The individuals obtained as results of the evolutionary algorithm contain all the information needed to configure the reference FPGA through the use of the XACT development system provided by Xilinx. Configuration is obtained by specifying user-designated partitioning and placement as part of the design entry process [14].

3.1 FPGA Reference Architecture

We consider the FPGA device XC3020, made of an array of 8×8 configurable logic blocks (CLBs) [15]. Each CLB has one output bit and four input bits. More recent FPGA families (e.g. XC4000) offer more design flexibility, both in the number and in the complexity of blocks. However, the simplicity of the XC3020 makes it the ideal candidate for an evolutionary study. Moreover, later FPGA families are fully compatible with the XC3020 device.

Since the hashing function does not require any state information and therefore can be computed through a combinational circuit, the registers of the FPGA are not used. Our design employs only the combinational logic of each CLB, as illustrated in Fig. 1.

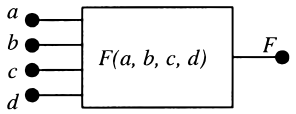


Fig. 1 The elementary cell of the circuit.

The complete combinational circuit is illustrated in Fig. 2. The output bits of the 8 rightmost blocks are the output of the circuit, i.e. the index computed by the hashing function.

We studied several different interconnection strategies for the circuit cells. Each of them is described by specifying the list of possible input lines for each cell, i.e. its *neighbourhood*.

The neighbourhoods considered in this paper are the following:

prev3 Each cell can receive the outputs of the three adjacent cells in the previous column; the cells in the first column can take two bits each from the input key.
prev5 Each cell can receive the outputs of three cells in the previous column, from the two rows above and from the second row below. Furthermore, each cell can receive two bits of the input key, which are distributed by row, so that bits $2i$ and $2i + 1$ are available to cells in row i.
gamma Each cell can receive the outputs of the three adjacent cells in the previous column and of the cell above in the same column, as well as two bits of the input key, which are distributed as in topology *prev3*.
minimal Only the last column is used: the eight cells compute one bit of the output index using any of the 16 bits of the input key without communicating with each other.
stripes For each cell, inputs can be any bit of the input key or the output of a previous cell located in the same row. In this way, each row processes the same input data independently.
layer2 Only columns 5, 6 and 7 are used; cells in columns 5 and 6 receive the external input lines, while cells in column 7 receive the outputs of the other cells.

Of course, these six interconnection strategies do not cover all possible topologies; they were chosen according to the following criteria:

input propagation: each input signal should be able to affect any of the output lines;
interconnection modularity: all cells, except those on the border, should have the same neighborhood topology;
locality: maximal use should be made of *direct interconnects* [14] between adjacent CLBs;

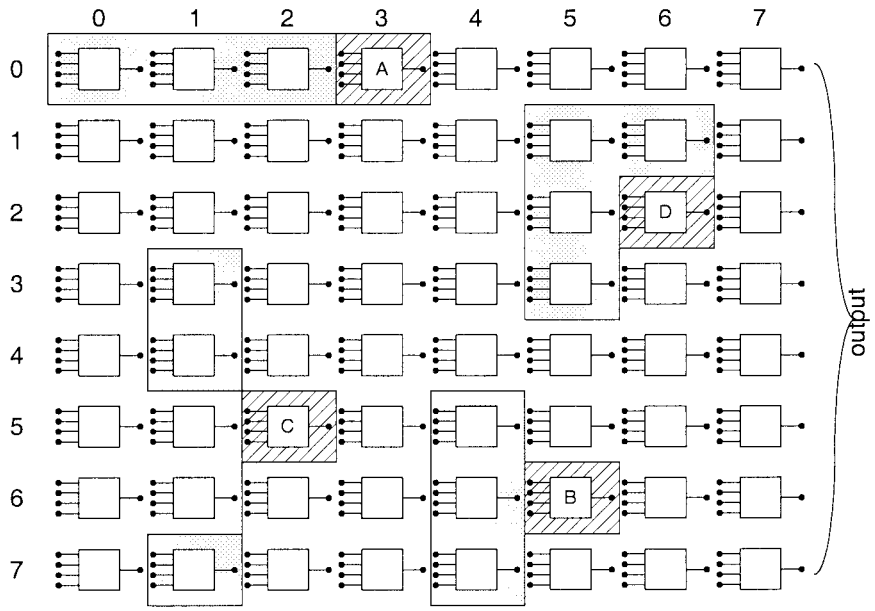


Fig. 2 The reference architecture adopted for the hashing circuits and four different neighbourhoods: (A): *stripes*; (B): *prev3*; (C): *prev5*; (D): *gamma*.

incremental complexity: the proposed interconnection strategies should allow a range of increasing complexity.

Moreover, the chosen strategies are significantly different from one another.

The first four connection strategies are guaranteed to be safe. In strategies *prev5* and *gamma*, the 16 external lines are driven horizontally or vertically, one for each row and one for each column (*gamma*) or horizontally, two for each row (*prev5*). Such input lines are available to all cells of the corresponding row or column. This interconnection of external inputs exploits the routing facilities available in the FPGA.

The fifth strategy (*stripes*) was already studied in a previous work [16] and has been included for completeness. Although some of its layouts might be non routable, it turned out that most of those found by the evolutionary algorithm were in fact routable.

The sixth strategy (*layer2*) mimics the structure of a 2-layer neural network. Although it is not safe from the routing point of view, it has been considered for comparison purposes. On the other hand, *prev3* is a safe version of the same concept, as external inputs can be read only by cells in the first layer, and then signals flow from layer to layer, even though with some connectivity restrictions.

Four of the six connection strategies are illustrated in Fig. 2.

In all possible circuits, the signal flow is from left to right (and from top to bottom in *gamma*), and there is no feedback loop.

3.2 Behavioural Description

The modularity of the circuit allows us to specify the behaviour simply by describing each CLB. For each block, we must specify its four inputs (which can come from input lines, from other cells, or can be grounded), and how they are combined to give the output. No global signal or supervision mechanism is used.

The algorithm reads the neighbourhood graph, which defines the possible inputs for each cell of the circuit. The connectivity map is then obtained through evolution, according to the constraints imposed by the graph.

The neighbourhood graph affects the routability of the circuit. Generally speaking, the larger the number of possible inputs for each cell, the higher the probability of exceeding the routing resources of the FPGA. Some graphs, however, can be considered as "safe" from the point of view of the routability, because all possible circuits conforming to such connection rules require a number of lines compatible with the hardware resources.

4. The Evolutionary Algorithm

In order to make hashing circuits evolve, a very simple generational evolutionary algorithm using fitness-proportionate selection with elitism and linear scaling of the fitness has been implemented.

```
SeedPopulation(popSize)
generation := 0
while true do
    for i := 1 to popSize do
        EvaluateFitness(i)
    end for
    Selection
    for i := 1 to popSize step 2 do
        Crossover(i, i + 1, p_cross)
    end for
    for i := 1 to popSize do
        Mutation(i, p_mut)
    end for
    generation := generation + 1
end while
```

Fig. 3 The pseudo-code of the evolutionary algorithm.

The overall flow of the evolutionary algorithm, which operates on an array *individual*[*popSize*] of individuals (i.e. the population) is illustrated in Fig. 3; even though this is not explicitly demonstrated in the pseudo-code for sake of readability, crossover and mutation are never applied to the best individual in the population.

The various elements of the algorithm are illustrated in the following subsections.

4.1 Encoding

An encoding was adopted that satisfies the constraints described in Sect. 3 by construction. Each cell is encoded through five 16-bit unsigned integers; the first four of them define the four inputs to the cell, which can be either input lines of the circuit or outputs of previous cells in the same row. The fifth integer encodes the truth table of the combinational network the cell implements.

Each of the integers encoding for the cell inputs is decoded by taking the rest of the division of its value by the number of legal inputs to the cell, depending on the adjacency matrix defining the topology and connectivity of the cells, plus the ground line.

In the case of the *stripes* connectivity scheme described in Sect. 3.1, the possible legal inputs to the cell are the sixteen inputs lines of the circuit, the output lines of the previous cells in the same row and the ground line. This gives 17 possibilities for the first cell up to 24 for the last cell in a row.

The null (ground) input allows the algorithm to evolve the number of inputs to each cell, as well as its logical function, while maintaining a fixed-length encoding.

As an example, the cell in Fig. 4 could be encoded as follows:

$$(400, 821, 37, 399, 54990). \tag{1}$$

The reader should be aware that in general there are many alternative ways of encoding for the same cell. For instance, in this case there are 256 distinct ways of encoding just the same logical function, since one of the four inputs has been grounded.

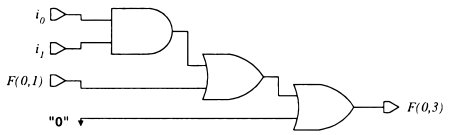


Fig. 4 Equivalent circuit diagram of a sample cell, imagined to be located in $(0, 3)$, as decoded from Expression 1.

Overall, a circuit is represented by a string of 5,120 bits, interpreted as a vector of $64 \times 5 = 320$ 16-bit integers. This is the genotype corresponding to a candidate solution.

4.2 Mutation

Whenever an individual is replicated to be inserted in the new generation, each bit of its genotype is flipped with the same probability p_{mut} and independently of the others.

There are some considerations to be done with respect to the encoding of cells:

1. for the truth table part, this mutation induces a small perturbation that does not disrupt the semantics of the genotype;
2. for the input selection part, flipping one or more bits amounts to replacing the old input with a new one at random, at least to a first approximation. In fact, it could be shown that codes corresponding to least significant input lines have a slightly higher probability of being generated. However, the bias introduced is negligible.

The mutation rate has to be chosen carefully, since it can dramatically affect the overall algorithm performance. Indeed, if the mutation rate is too low, the evolution is exceedingly slow; if it is too high, the algorithm tends to behave as a random search.

We have experimentally verified that the algorithm performance is good when the ratio $r = f^*/\bar{f}$, where f^* is the fitness of the best individual and $\bar{f}$ is the average fitness of the population, is close to $R = 3/2$.

The mutation rate p_{mut} is dynamically adjusted in order to maintain the ratio r as close as possible to its optimal value R. The updating rule is:

$$p_{\text{mut}} \leftarrow \frac{R}{r} p_{\text{mut}}. \tag{2}$$

If the distance between the best and the average circuit increases, p_{mut} is accordingly lowered, while if the average circuit is catching up with the best, p_{mut} is increased to allow for more diversity in the population.

4.3 Recombination

For recombination, each 16-bit unsigned integer is treated as an atomic unit of the genotype.

Each such unit is inherited from either parent with 1/2 probability (uniform crossover). This ensures that recombination will preserve the CLBs, which are the basic building blocks of a circuit.

4.4 Objective Function and Fitness

A "good" hashing function should map a set of M input keys into a set of indices $i = 1, \ldots, N$ in the most uniform way as possible. In order to assess the quality of the generated circuits, a set of keys is randomly generated over the set $\{1, \ldots, 2^{16}\}$ of possible key values. These keys are offered as inputs to each circuit and the number of hits h_i for each index i is calculated. The fitness of a circuit is then given by the following formula:

$$f = \frac{1}{1 + \frac{1}{N}\sum_{i=1}^{N}(h_i - \bar{h})^2}, \tag{3}$$

where $\bar{h} = M/N$ is the expected number of hits per index in the ideal case of uniform key distribution.

An alternative measure of fitness could be based on the entropy of the key distribution generated by the circuit, i.e.

$$f = MH = \sum_{i=1}^{N} h_i \ln \frac{M}{h_i} = M \ln M - \sum_{i=1}^{N} h_i \ln h_i. \tag{4}$$

However, it was experimentally verified that such a fitness reduces the algorithm performance, for it does not sufficiently reward the best circuits, and the additional reward for a better circuit greatly diminishes as a high-entropy solution is approached.

In order to evaluate the results, it is interesting to study the probabilistic behaviour of a "random" hashing function. By "random" we mean that the probability that a random key is mapped to any index is independent of the index and uniform over the N indices. The number of hits for all indices $i = 1, \ldots, N$, would have binomial distribution with probability

$$\Pr[h_i = h] = \binom{M}{h} \frac{(N-1)^{M-h}}{N^M}. \tag{5}$$

Therefore, the mean is

$$E[h_i] = \bar{h} = \frac{M}{N} \tag{6}$$

and the variance is

$$\mathrm{var}[h_i] = \frac{M(N-1)}{N^2}. \tag{7}$$

5. Experimental Results

Experiments have been carried out with $N = 256$ and $M = 4,096$ out of the 65,536 possible keys. This gives an average number of hits per index $\bar{h} = 16$, regardless of the hashing function employed.

A population size of 100 was used, with crossover rate $p_{\mathrm{cross}} = 1/2$, unless otherwise specified.

According to (3), the standard deviation σ of hits per index for a hashing circuit having fitness f is

$$\sigma = \sqrt{\frac{1}{f} - 1}. \tag{8}$$

For the random hashing function, the distribution of hits per index given in (5) would have mean $\bar{h} = 16$ and variance 15.9375, or standard deviation $\sigma_{\mathrm{rnd}} = 3.9922$. This can serve as a term for comparison to assess the effectiveness of the evolved solutions.

Figure 5 shows a typical circuit synthesized by the algorithm described in Sect. 4 with the *stripe* topology. It was obtained after only 257 generations, yet it generates a key distribution with a standard deviation of $\sigma = 3.0375$ hits per index, which is considerably better than the σ_{rnd} of the random hashing function. This means that the evolved circuit is actually fitted to the actual set of keys used for evaluating fitness.

It has been observed that evolution proceeds in three successive phases:

1. in the first few generations, the best combinational functions are selected for blocks;
2. in the second phase, the topology of circuits evolves in such a way that each input bit has a path through the output;
3. in the last phase, the evolution process is slower, and tends to increase the connectivity within circuit blocks.

As shown in Fig. 6, the first phase is characterised by a steep increase of fitness and by a high rate of change in the population. In subsequent phases, one can observe the typical punctuated equilibria where abrupt changes are interleaved with long periods of stasis.

The results for the *prev3* connectivity scheme are shown in Table 1. It can be observed that they are disappointingly poor. For this reason the runs were terminated at generation 450.

The results for the *prev5* connectivity scheme are shown in Table 2.

It is interesting to compare the four runs of *prev5* at the same generation, namely generation 508: the best of those with $p_{\mathrm{cross}} = 0.5$ had already attained the respectable fitness of 0.112084 whereas that with $p_{\mathrm{cross}} = 0.3$ had 0.090845, *vis-à-vis* the 0.099922 obtained by that with $p_{\mathrm{cross}} = 0.7$. This is a confirmation that a crossover rate of about 1/2 is optimal, but

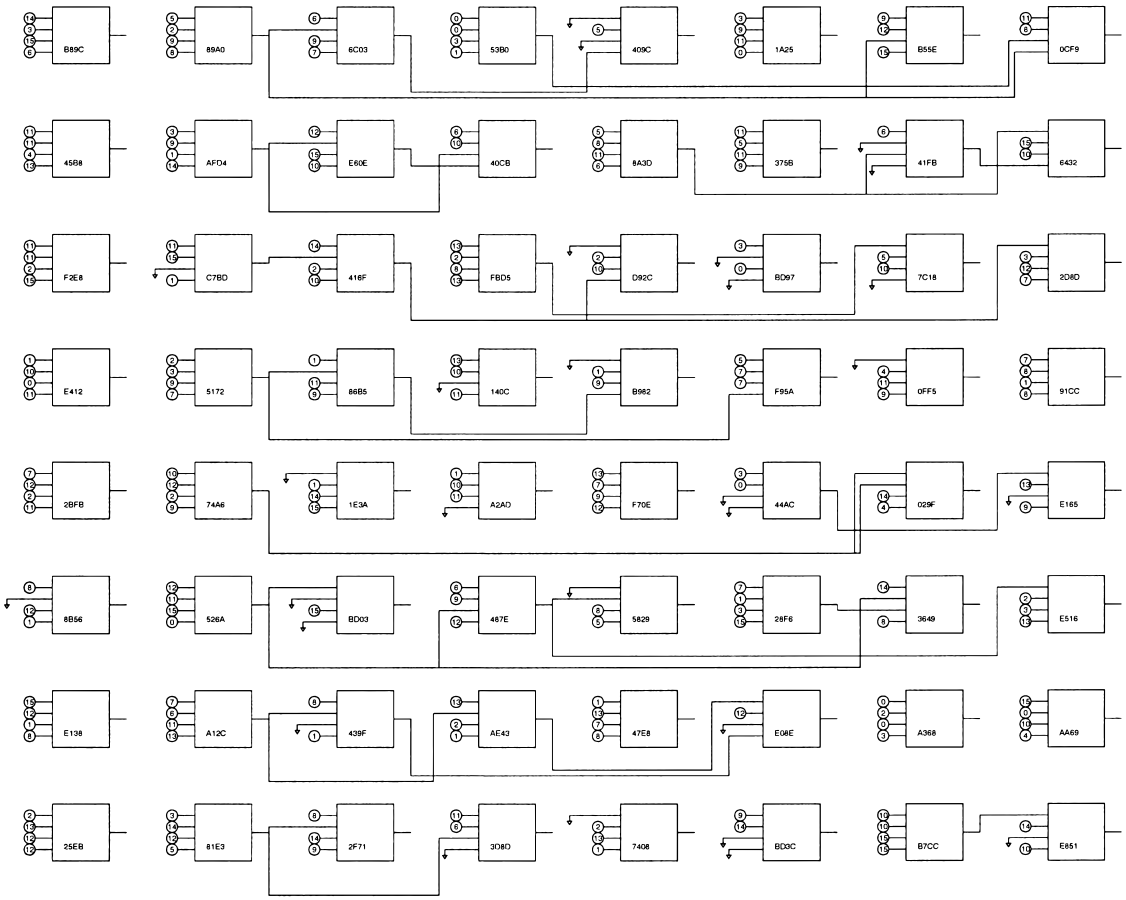


Fig. 5 The best circuit for the *stripes* topology at Generation 257, having fitness 0.097785.

Table 1 Fitness of the best individual after 450 generations in three runs of the evolutionary algorithm with the *prev3* topology.

Run	Fitness at generation 450
1	0.000888
2	0.000859
3	0.001879

Table 2 Fitness of the best individual after four runs of the evolutionary algorithm with the *prev5* topology.

No. of generations	Best fitness	Note
4919	0.109215	
3163	0.129162	
3272	0.122841	$p_{\text{cross}} = 0.3$
508	0.099922	$p_{\text{cross}} = 0.7$

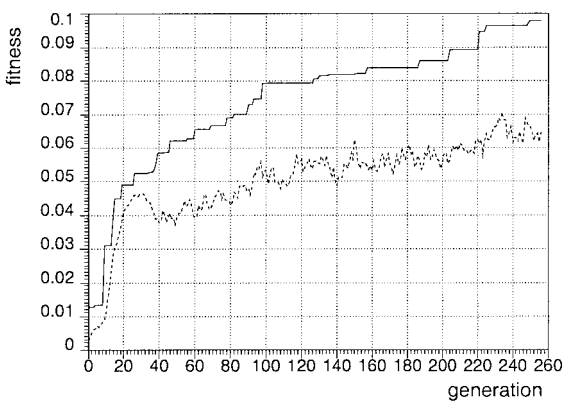


Fig. 6 Average and best fitness during a sample run of the evolutionary algorithm.

indicates that higher mutation rates lead to a slower decrease in terms of performance.

Table 3 compares the fitness of the best individual obtained in four distinct runs of the algorithm with the *gamma* connectivity scheme at the same generation: it can be observed that after 4,500 generations the average fitness of the best individual is 0.121821, with a standard deviation of 0.008124, very small indeed.

This topology was able to generate the best hash circuits with respect to the criteria adopted. For this reason, we allowed some of the runs in Table 3 to continue for a large number of generations, obtaining an excellent circuit having fitness 0.139738 after 13,911 generations. The scheme of this circuit is shown in Fig. 7. It produces a key distribution with a standard deviation of $\sigma = 2.4812$ hits per index.

The *minimal* connectivity scheme was able to

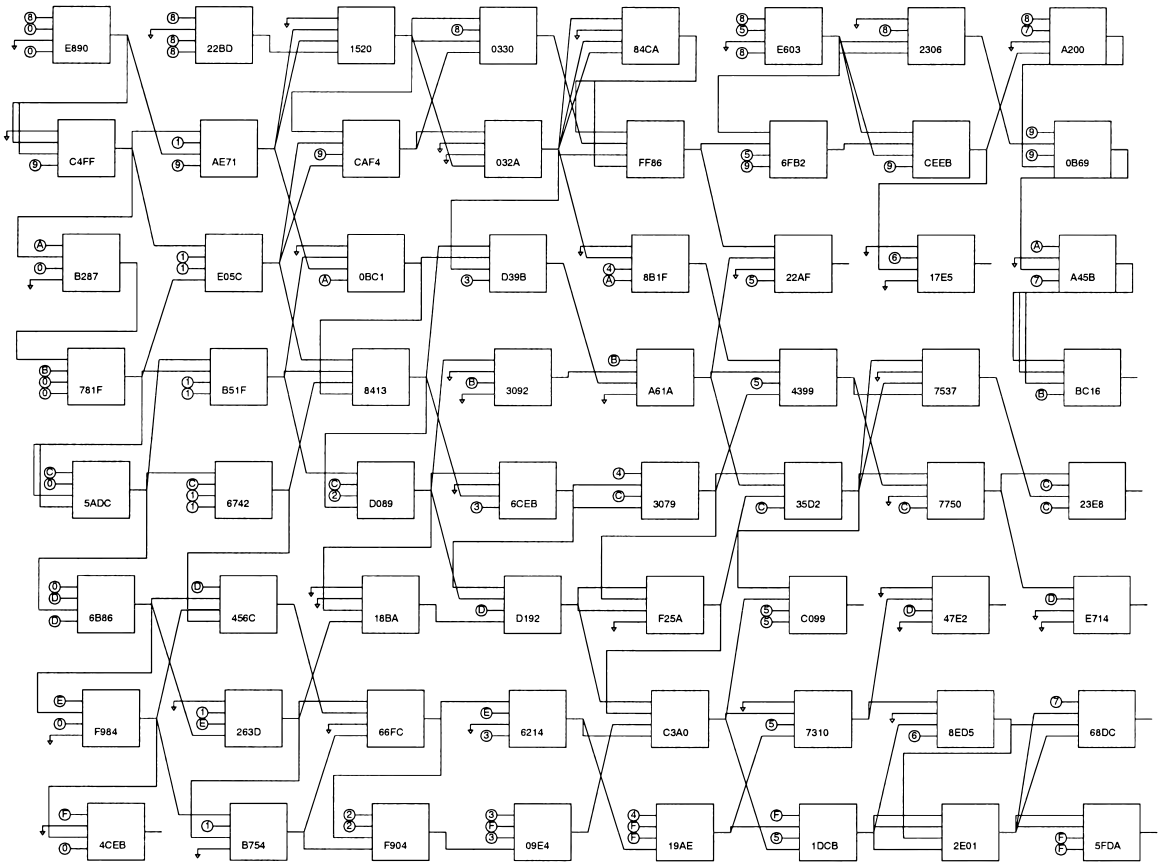


Fig. 7 The best circuit produced for the *gamma* topology at Generation 13,911, having fitness 0.139738.

Table 3 Fitness of the best individual after 4,500 generations in four runs of the evolutionary algorithm with the *gamma* topology.

Run	Fitness at generation 4,500
1	0.112182
2	0.131417
3	0.124393
4	0.119292

Table 4 Fitness of the best individual after 4,500 generations in two runs of the evolutionary algorithm with the *minimal* topology.

Run	Fitness at generation 4,500
1	0.100708
2	0.093979

reach a fitness level of about 0.1, but there it got stuck in all experiments. Results for two sample runs are shown in Table 4.

The results for the *layer2* connectivity scheme are shown in Table 5. Only two runs are reported because it was rather clear from the beginning that this connectivity scheme shows limited room for improvement.

6. Discussion of Results

The connection strategies described in Sect. 3 can be compared from several points of view, namely

Table 5 Fitness of the best individual after 4,500 generations in two runs of the evolutionary algorithm with the *layer2* topology.

Run	Fitness at generation 4,500
1	0.095309
2	0.093636

- the fan-in of each node in the neighbourhood graph,
- the minimum length of a path from an input to an output node,
- the number of connected nodes, i.e. of cells available for the computation.

First of all it can be observed that both the connectivity strategies with a very low or very high fan-in tend to perform rather poorly in terms of convergence speed of the evolutionary algorithm. This observation is important in view of on-line implementation of the proposed approach. A low connectivity, as in the *minimal* and *prev3* connection strategies, imposes too harsh restrictions on the search space. On the other hand, discovering good circuits when connectivity is high is like looking for a needle in a haystack, because the search space becomes indeed huge, apart from the fact that the probability of coming up with unroutable circuits becomes non-negligible.

From another perspective, one is struck by how the "layered" connection strategies, namely *layer2* and *prev3*, perform significantly worse even than the *minimal* connectivity, which can be regarded as a reference for comparison. This can be explained by the fact that every bit of the input key can be taken into account in the calculation of the output index only through a path of cells, each one performing some logical operation. Therefore, it is very easy for mutation or crossover to disrupt the operational semantics of a potentially good component. This *brittleness* of solutions is a very well known effect of inadequacy of encoding or genetic operators. As a consequence, while we do not rule out that good circuits might be found using those connection strategies, it is possible that a higher-level type of encoding or more semantics-aware operators would have to be used.

The two connection strategies that gave the best results, with respect to both convergence speed and quality of the final results, *prev5* and *gamma*, seem to strike a balance between the two extremes of excessive simplicity and unmanageable complexity. Both make use of the full set of available cells. Furthermore, they make external inputs available to entire rows or columns, thus providing to evolution many ways of leading a certain signal to any of the output cells. Finally, they both have the desirable property of being safe from the routability point of view.

We can then safely conclude that connection strategies of this kind are the best candidates for evolutionary synthesis of FPGA configurations, in particular if on-line evolution is required.

7. Applications and Future Work

This Section outlines a possible application for the evolutionary technique described in this paper: the adaptive design of cache memories. Commonly available cache memories are either *fully* or *set-associative* [17]. A fully associative cache memory comprises several *block frames* holding copies of main memory blocks. The cache also contains a table, holding the address of the main memory block associated to each block frame.

If a linear 32-bit address space is used for main memory and a block size of 64 kbyte is adopted, the table holds a set of 16-bit block addresses or *tags*, each coupled with the corresponding block frame identifier. Fully associative memories operate as follows: before the CPU performs an access to main memory, a fast combinational network simultaneously compares the 16 most significant bits of the desired address with all the block tags stored in the table. If a matching is found, the cache block frame is accessed instead of the main memory block. Otherwise, main memory is accessed and the missing block is copied in cache. The capacity of fully associative cache memories is severely limited by the topological complexity of the combinational network used to perform simultaneous comparisons. For this reason, large cache memories used in current microcomputer architectures are usually set-associative. A typical set-associative cache maps 16-bit block identifiers to a smaller set of 256 tags. This is usually done by performing associative comparison on the most significant 8 bits of the block frame identifier to select a set of blocks. The desired block is then located via linear search, usually without excessive performance degradation [18].

The circuit described in previous sections can be readily used to achieve uniform mapping of block identifiers to tags, as an alternative to associative comparison. However, it is interesting to remark that if actual memory accesses are not evenly distributed across blocks, as it is often the case, uniform mapping is not the best choice for set-associative cache memories. Indeed, the cost of linear search would be decreased by letting several seldom consulted blocks to share a tag and countersigning frequently accessed blocks with a unique tag. The "best" mapping between tags and block sets should be dynamically searched while the circuit operates, through a second evolutionary phase of *on-line* optimisation. We intend to explore this issue in a future paper.

8. Conclusion

This paper has described the evolutionary design of a digital circuit to compute a non-linear mapping suitable for hashing and associative memories. The feasibility of the approach has been demonstrated. Further research is in progress, in order a full implementation in hardware of the described algorithm, much in the spirit of the evolvable hardware approach.

Acknowledgments

This work was partially supported by M.U.R.S.T. 60% funds.

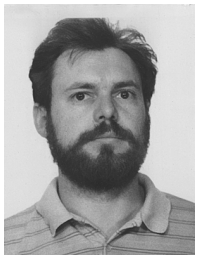
The authors would like to thank Prof. Gianni Degli Antoni for his support and valuable criticism and Prof. Nello Scarabottolo for his advice on hardware applications.

References

[1] C. Darwin, On the Origin of Species by Means of Natural Selection, John Murray, London, UK, 1859.
[2] R. Dawkins, The Blind Watchmaker, Norton, New York, NY, USA, 1987.
[3] J.H. Holland, Adaptation in Natural and Artificial Systems, The University of Michigan Press, Ann Arbor, MI, USA, 1975.
[4] L.J. Fogel, A.J. Owens, and M.J. Walsh, Artificial Intelligence through Simulated Evolution, John Wiley & Sons, New York, NY, USA, 1966.
[5] I. Rechenberg, "Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution,"

Fromman-Holzboog Verlag, Stuttgart, Germany, 1973.
[6] J.R. Koza, Genetic Programming: On the Programming of Computers by Means of Natural Selection, The MIT Press, Cambridge, MA, USA, 1993.
[7] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, Springer-Verlag, Berlin, Germany, 1992.
[8] T. Bäck, Evolutionary Algorithms in Theory and Practice, Oxford University Press, Oxford, UK, 1996.
[9] R. Drechsler, Evolutionary Algorithms for VLSI CAD, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1998.
[10] M. Sipper, E. Sanchez, D. Mange, M. Tomassini, A. Pérez-Uribe, and A. Stauffer, "A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems," IEEE Trans. Evolutionary Computation, vol.1, no.1, pp.1–14, Feb. 1997.
[11] J.F. Miller, P. Thomson, and T. Fogarty, "Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study," in Genetic Algorithms and Evolution Strategies in Engineering and Computer Science, eds. D. Quagliarella, J. Périaux, C. Poloni and G. Winter, John Wiley & Sons, New York, NY, USA, 1998.
[12] M. Tanaka, H. Sakanashi, M. Salami, M. Iwata, T. Kurita, and T. Higuchi, "Data compression for digital color electrophotographic printer with evolvable hardware," in Proc. Second International Conference on Evolvable Systems (ICES '98), eds. M. Sipper, D. Mange and A. Pérez-Uribe, pp.106–114, Lausanne, Switzerland, Sept. 1998.
[13] J.R. Koza, F. Bennett, D. Andre, and M. Keane, "Evolution using genetic programming of a low distortion 96 decibel operational amplifier," Proc. ACM Symp. on Applied Computing (SAC '97), San José, CA, 1997.
[14] Xilinx, Inc., San José, CA, The Programmable Logic Data Book, 1996.
[15] J.H. Jenkins, Designing with FPGAs and CPLDs, Prentice-Hall, Englewood Cliffs, NJ, USA, 1994.
[16] E. Damiani, V. Liberali, and A.G.B. Tettmanzi, "Evolutionary design of hashing function circuits using an FPGA," in Proc. Second International Conference on Evolvable Systems (ICES '98), eds. M. Sipper, D. Mange and A. Pérez-Uribe, pp.36–46, Lausanne, Switzerland, Sept. 1998.
[17] A. Smith, "Cache memory design: An art evolves," IEEE Spectrum, vol.24, 1987.
[18] W. Burkardt, "Locality aspects and cache memory utility in microcomputers," Euromicro Journal, vol.26, 1989.

Ernesto Damiani received a Master degree in electronic engineering in 1987 from the University of Pavia and a Ph.D. in computer science in 1994 from the University of Milan. Since 1995 he is an assistant professor in computer science at the University of Milan. He is also a visiting professor at the Computer Science Department of La Trobe University, Melbourne, Australia. He is the author of more than 40 scholarly publications in the field of distributed and object-oriented systems, fuzzy logic and soft computing.



Valentino Liberali received a Master degree in electronic engineering in 1986 from the University of Pavia. Since 1990, he is an assistant professor in electronics at the University of Pavia. He is the author of more than 40 scholarly publications in the field of analog and mixed-signal CMOS integrated circuits and electronic CAD.

Andrea G. B. Tettamanzi received a Master degree in computer science in 1991 and a Ph.D. in computational mathematics and operations research in 1995 from the University of Milan. From 1994 to 1995 he was with the Computer Science Division of the University of California, Berkeley, as a visiting scholar. In 1995 he established Genetica—Advanced Software Architectures S.r.l. (a consulting firm in computer science applications) in Milan, where he directs the R&D department. Since 1998 he is an assistant professor in computer science at the University of Milan. He is the author of several scholarly publications in the field of evolutionary algorithms, fuzzy logic and soft computing.