Evolutionary Design of Hashing Function Circuits using an FPGA

Ernesto Damiani[1], Valentino Liberali[2], and Andrea G. B. Tettamanzi[1]

[1] Università degli Studi di Milano, Polo Didattico e di Ricerca di Crema
Via Bramante 65, 26013 Crema, Italy
Phone: +39-373-898201; Fax: +39-373-898253
E-mail: edamiani@crema.unimi.it, tettaman@dsi.unimi.it
[2] Università di Pavia, Dipartimento di Elettronica
Via Ferrata 1, 27100 Pavia, Italy
Phone: +39-382-505226; Fax: +39-382-505677
E-mail: valent@ele.unipv.it

Abstract. 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. This circuit, based on FPGAs, is readily applicable to the design of *set-associative* cache memories. Possible use the evolutionary approach presented in the paper for *on-line* tuning of the function during cache operation is also discussed.

1 Introduction and 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 digital 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.

Evolutionary algorithms are a broad class of optimization methods inspired by Biology, that build on the key concept of Darwinian evolution [3, 5]. After some successful applications of evolutionary algorithms to physical design (partitioning, placement and routing), now the same techniques are being considered also for structural design. This has given rise to an innovative field of research, called *evolvable hardware* [8, 19, 17]. Recently, evolutionary design automation has been applied to digital electronic design [14]. It has been proposed for analog circuits as well, with encouraging results [12].

The main concepts and issues relevant to evolutionary algorithms can be found, among others, in [16, 7, 4, 13, 11, 1]; [9, 6, 15] are more of a historical interest.

M. Sipper et al (Eds.): ICES'98, LNCS 1478, pp. 36-46, 1998.
© Springer-Verlag Berlin Heidelberg 1998

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

2 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 [14].

The Xilinx SRAM-based reconfigurable FPGAs have been chosen as the reference architecture for this application [20].

2.1 Architecture of FPGA Implementation

We consider the FPGA device XC3020, made of an array of 8×8 configurable logic blocks (CLBs) [10]. 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 this FPGA 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 Figure 1.

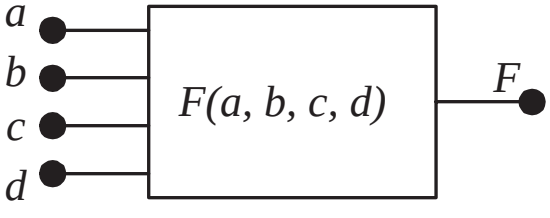

Fig. 1. The elementary cell of the circuit.

The complete combinational circuit illustrated in Figure 2. For each CLB, each inputs can be any bit of the input key or can be the output of a previous

cell located in the same row. In this way, each row processes the same input data independently. The signal flow is from left to right, and there is no feedback loop. The 16 inputs are transported on the horizontal long lines and distributed to the cells using both vertical general purpose interconnects and long lines. Connections between cells within the same row are carried out by horizontal general purpose interconnect. These conventions do not guarantee that all individuals generated by the algorithm will be routable using the FPGA hardware resources; however, in practice the individuals that cannot be routed using this technique are effectively filtered out by assigning zero fitness to them.

The output bits of the 8 rightmost blocks are the output of the circuit, i.e. the index computed by the hashing function.

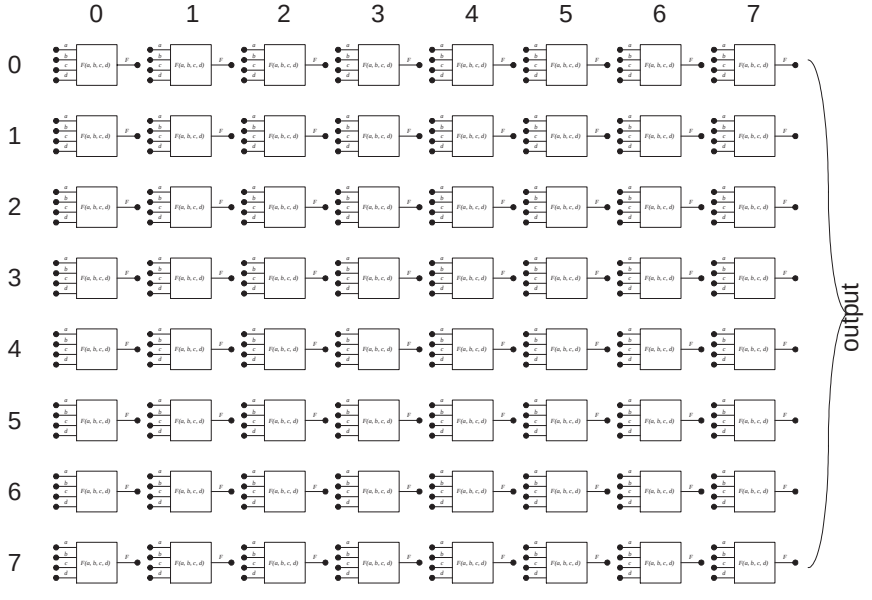

Fig. 2. The reference architecture adopted for the hashing circuits.

2.2 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 previous cells, or can be grounded (i. e. bounded to zero), and how they are combined to give the output.

No global signal or supervision mechanism is used.

3 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, whose details are given in the following subsections.

3.1 Encoding

An encoding was adopted that satisfies the constraints described in Section 2 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. Even though five bits per input would be enough for coding 24 possible inputs, the choice of using a full 16-bit word is dictated by performance considerations, all the more so because the genotype already fitts in less than 5 kbytes of memory, which is negligible for state-of-the-art computers. 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, i.e. 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 Figure 3 could be encoded as follows:

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

The first four numbers encode the four inputs. Being the cell in column 3, its inputs can come from 20 possible sources, namely, one of the 16 inputs, outputs of cells (0,0), (0, 1), (0,2), and the ground. Thus, $400(\mod 20) = 0$, indicating i_0 as the first input; $821(\mod 20) = 1$, yielding i_1, and so on. The last number, $54990 = (1101011011001110)_2$, is a straightforward encoding of the truth table defining $F(0, 3)$. 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.

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.

3.2 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.

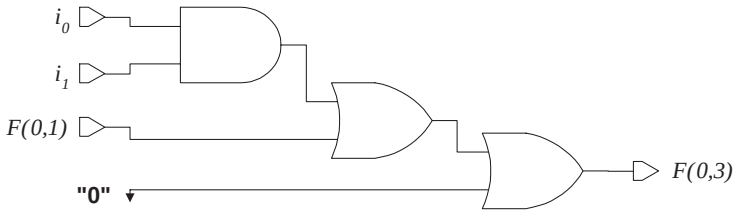

Fig. 3. Equivalent circuit diagram of a sample cell, imagined to be located in $(0, 3)$, as decoded from Expression 1.

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 in that flipping a bit in this part amounts to modifying a single entry of the truth table; the resulting function is identical to the former one except for the value associated to a single input;
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;
3. thanks to the design of the encoding, the offspring resulting from mutation are always legal; at most, they will not be routable (and therefore unfit for reproduction).

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 $R = \frac{3}{2}$.

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:

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

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.

3.3 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 $\frac{1}{2}$ probability (uniform crossover). This ensures that recombination will preserve the CLBs, which are the basic building blocks of a circuit.

3.4 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 $E[h_i] = \bar{h} = M/N$ and the variance is $\mathrm{var}[h_i] = \frac{M(N-1)}{N^2}$.

4 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 $p_{\text{cross}} = \frac{1}{2}$.

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

$$\sigma = \sqrt{\frac{1}{f} - 1}. \tag{6}$$

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 $\sigma_{\text{rnd}} = 3.9922$. This can serve as a term for comparison to assess the effectiveness of the evolved solutions.

Figure 4 shows a typical circuit synthesized by the algorithm described in Section 3. It has been obtained after 257 generations, and 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. Repetition of the experiments always leads to similar results. This means that the evolved circuit is actually fitted to the actual set of keys used for evaluating fitness.

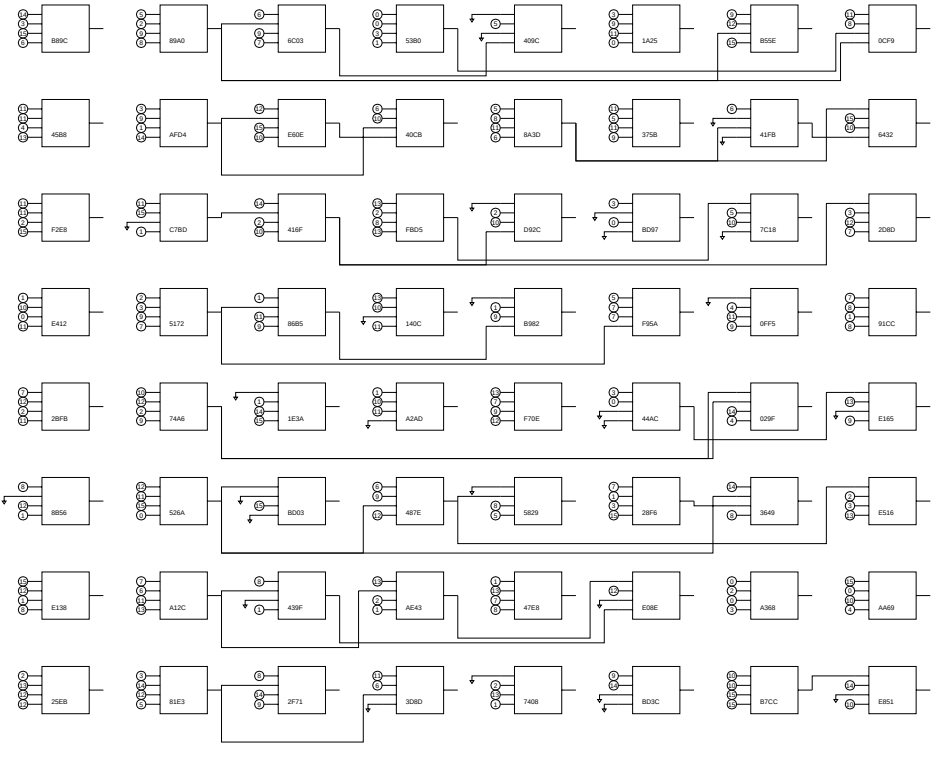

Fig. 4. The best circuit at Generation 257, having fitness 0.097785.

It has been observed that the evolution proceeds in three successive phas s:

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

As shown in Figure 5, the first phase is characterised by a steep increase of fitn and by a high rate of change in the population. In subsequent phases, one c observe the typical punctuated equilibria where abrupt changes are interleav with long periods of stasis.

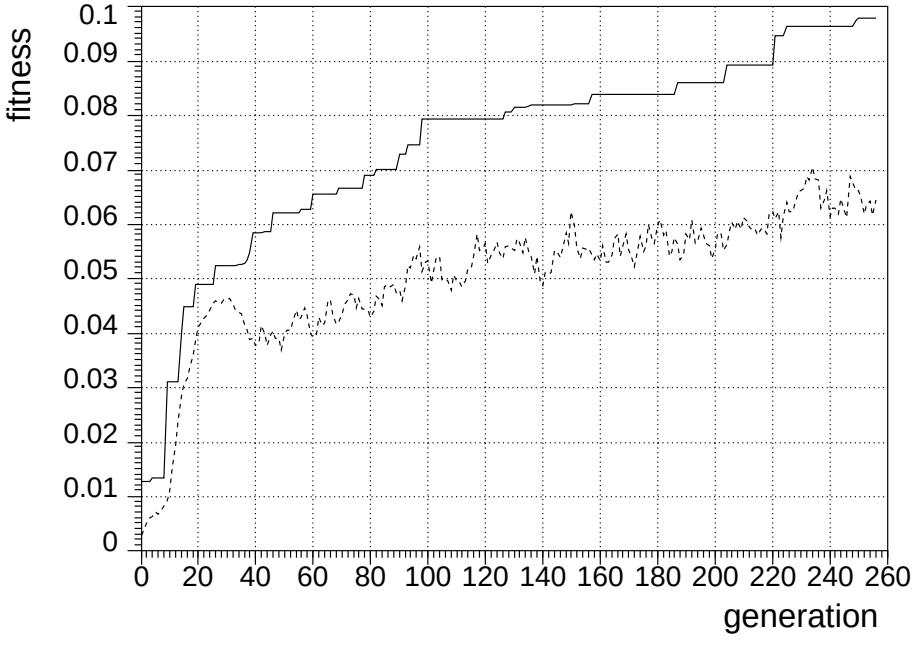

Fig. 5. Average and best fitness during a sample run of the evolutionary algorithm.

5 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* [18]. 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 [2].

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. In this case, each identifier can be regarded as an hash key identifying a set of 256 cache blocks which must be searched in order to locate the desired one. 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 of blocks into tags is not the best choice for set-associative cache memories. Indeed, the cost of linear search would be decreased by letting 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.

6 Discussion

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 to obtain a full implementation in hardware of the described algorithm, much in the spirit of the evolvable hardware approach.

The circuit layout proposed above has some advantages, above all, a regular structure allowing a straightforward encoding. However, it is neither the only possible, nor perhaps the best. As a matter of fact, it could be interesting to allow the connections between cells in a column-by-column fashion, instead of a within-row fashion. Furthermore, since the solutions obtained in Section 4

correspond to a small depth functions, smaller portions of the 8×8 array could be used to implement the solution. By the way, this approach would lead to an overall layout reminiscent of multi-layer perceptrons.

Another interesting possibility (which has not been yet taken advantage of) is to use an actual FPGA for fitness evaluation. Given that each fitness evaluation implies running M software simulations of the same circuit, it is evident that hardware fitness evaluation becomes more and more attractive as M increases.

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 criticism. We are also grateful to the anonymous referees for their valuable suggestions and observations.

References

1. T. Bäck. *Evolutionary algorithms in theory and practice.* Oxford University Press, Oxford, UK, 1996.
2. W. Burkardt. Locality aspects and cache memory utility in microcomputers, *Euromicro Journal*, vol. 26, 1989.
3. C. Darwin. *On the Origin of Species by Means of Natural Selection.* John Murray, 1859.
4. L. Davis. *Handbook of Genetic Algorithms.* VNR Computer Library. Van Nostrand Reinhold, New York, NY, 1991.
5. R. Dawkins. *The blind Watchmaker.* Norton, 1987.
6. L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution.* John Wiley & Sons, New York, NY, 1966.
7. D. E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning.* Addison-Wesley, Reading, MA, 1989.
8. T. Higuchi, T. Niwa, T. Tanaka, H. Iba, H. De Garis, T. Furuya. Evolving hardware with genetic learning: a first step towards building a Darwin machine. In *Proc. of the 2nd Int. Conf. on the Simulation of Adaptive Behavior (SAB92).* The MIT Press, Cambridge, MA, 1993.
9. J. H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, MI, 1975.
10. J. H. Jenkins. *Designing with FPGAs and CPLDs.* Prentice-Hall, Englewood Cliffs, NJ, 1993.
11. J. R. Koza. *Genetic Programming: on the programming of computers by means of natural selection.* The MIT Press, Cambridge, MA, 1993.
12. J. Koza, F. Bennett, D. Andre, M. Keane. Evolution using genetic programming of a low distortion 96 decibel operational amplifier, in *Proc. ACM Symp. on Applied Computing (SAC '97)*, San José, CA, Feb.-March 1997.
13. Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer-Verlag, Berlin, 1992.
14. J. F. Miller, P. Thomson, T. Fogarty. Designing electronic circuits using evolutionary algorithms. Arithmetic circuits: a case study, in D. Quagliarella, J. Périaux, C. Poloni, G. Winter (Eds.), *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science.* Wiley, Chichester, UK, 1998.

15. I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Fromman-Holzboog Verlag, Stuttgart, 1973.
16. H.-P. Schwefel. *Numerical optimization of computer models.* Wiley, Chichester, UK, 1981.
17. M. Sipper, E. Sanchez, D. Mange, M. Tomassini, A. Pérez-Uribe, A. Stauffer. A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems, *IEEE Trans. Evolutionary Computation*, vol. 1, 1997.
18. A. Smith. Cache memory design: an art evolves, *IEEE Spectrum*, vol. 24, 1987.
19. A. Thompson. Silicon evolution, in *Proc. of Genetic Programming 1996*, Palo Alto, CA, 1996.
20. Xilinx, Inc. *The Programmable Logic Data Book.* San José, CA, 1996.