Dynamic Optimisation of Non-Linear Feed-Forward Circuits

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
edamiani@crema.unimi.it, tettaman@dsi.unimi.it
[2] Università degli Studi di Pavia, Dipartimento di Elettronica
Via Ferrata 1, 27100 Pavia, Italy
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. The target technology is FPGA, where the search space of the algorithm is made of the combinational functions computed by cells, of the interconnection topologies and of the interconnections among cells. This circuit is readily applicable to the design of *set-associative* cache memories, with *on-line* tuning of the function during cache operation.

1 Introduction

Evolvable Hardware (EHW) is a relatively recent research field dealing with the synthesis of electronic circuits by means of evolutionary algorithms. The central idea of this approach is representing circuits as individuals in a evolutionary process in which the standard genetic operations can be carried out. At the first stages of EHW research, evolving circuits were often evaluated using software simulation models, while the final outcome had to be implemented in hardware by traditional methods. Today, thanks to the availability of *Field Programmable Gate Arrays (FPGA)*, the evolutionary process may be carried out *on-line*, periodically updating the FPGA structure and content to reflect ongoing evolution.

This work is aimed at on-line evolution of FPGA configurations in order to implement *adaptive hardware components*, i.e. subsystems that automatically modify their structure and operation at the gate and flip-flop level, according to the statistical properties of their input.

The conceptual architecture of an adaptive hardware component is shown in Fig. 1. It includes a *demultiplexer*, sending the input bytestream in parallel to a *working copy* of the adaptive component and to an *evolutionary engine*. The evolutionary engine contains a population of *candidate configurations* for the component and controls their evolution, computing their fitness with respect to the incoming data stream and, possibly, avoiding configurations that are illegal or undesirable for the target implemementation [1]. Functionally, the evolutionary engine comprises two subsystems: a *preprocessor* extracting from the input data

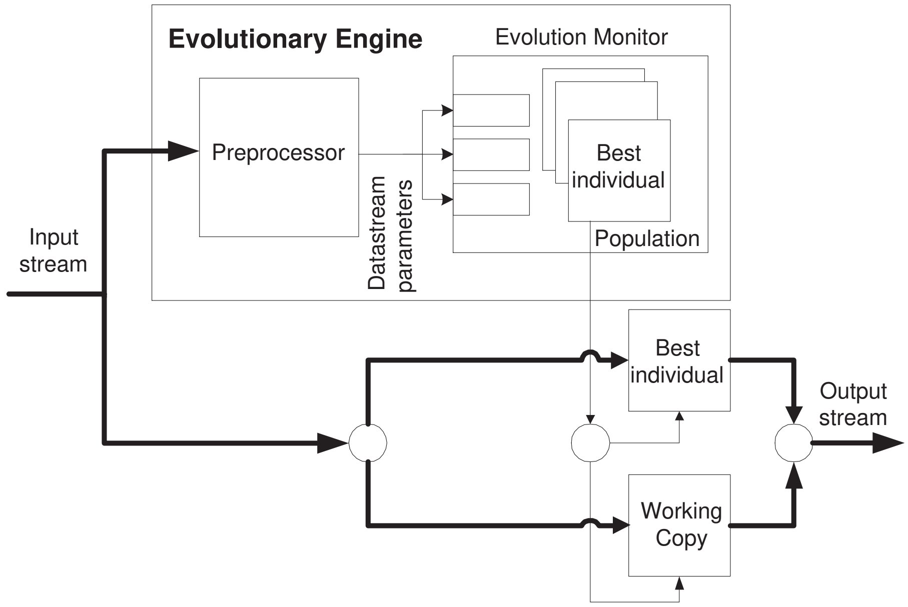

Fig. 1. Conceptual architecture of an adaptive hardware component

stream the parameters relevant to the computation of fitness, and the *evolution monitor* implementing the evolutionary algorithm. The evolutionary engine and the working copy of the circuit operate asynchronously, but the fitness of best individual of the population and the one of the working copy of the configuration are regularly compared. Every time the fitness of the best individual in the population overtakes the fitness of the working copy by a (tunable) threshold, the input data stream is switched from the working copy of the component to the best individual in the population, which becomes the new working copy. This conceptual architecture satisfies the requirement of ensuring uninterrupted operation of the evolvable component while being transparent both to the data and the control flow of the host system. Moreover, it can be instantiated to obtain several classes of components of practical interest. In the sequel, we shall focus on applying this approach to the evolution of a non-linear feed-forward circuit whose input is a stream of *keys*. The circuit computes a non-linear mapping of the keys to a smaller set of *indexes*, adapting the mapping to follow the dynamics of the keys' probability distribution.

2 Problem Formulation

Let $K = \{1, 2, \ldots, M\}$ be a set of keys to be mapped into a set $I = \{1, 2, \ldots, N\}$ of indices in the hash table, with $N \ll M$. Furthermore, assume keys in K have a probability distribution p_k, for all $k \in K$.

Let $\mathcal{H}\colon K \to I$ be the mapping calculated by a given hash circuit. Then, the probability distribution p_k on K is mapped by $\mathcal{H}$ to the associated probability distribution q_i on the set of indices such that, for all $i \in I$,

$$q_i = \sum_{k:\mathcal{H}(k)=i} p_k. \tag{1}$$

For each index $i \in I$, we can express the number of keys that $\mathcal{H}$ maps into i as $n_i = \sum_{k=1}^{M} [i = \mathcal{H}(k)]$, where notation $[P]$ stands for 1 if predicate P is true and 0 otherwise.

We are interested in finding a hash circuit that maps keys into indices of the hash table in such a way that the most probable (i.e. most frequent) keys have their own entries in the table, without collisions with other keys and that the number of other keys a given key has to share its entry with in the hash table be inversely proportional to its frequency.

The number of other keys a given key $k \in K$ shares an entry with is called the number of collisions c_k, given by $c_k = n_{\mathcal{H}(k)} - 1$. It can be observed that $c_k \geq 0$, because at least key k is mapped into $\mathcal{H}(k)$.

The design goal can be mathematically expressed by saying that we want to minimise the objective

$$z_{\mathcal{H}} = \sum_{k=1}^{M} p_k c_k^a, \tag{2}$$

where a indicates how much penalty we give to collisions as they stack up. For $a = 1$, the cost of each additional collision is constant, while for $a = 2$ it increases quadratically.

Assuming a linear penalty for additional collisions, it is possible to express the objective in Equation 2 in terms of features of the hash table, as follows:$z_{\mathcal{H}} = \sum_{k=1}^{M} p_k c_k = \sum_{i=1}^{N} q_i \sum_{k:\mathcal{H}(k)=i} c_k = \sum_{i=1}^{N} q_i n_i (n_i - 1)$.

In practical situations, we do not know the probability distribution over the set of keys and, therefore, we cannot calculate the q_i's. Besides, even though we could in principle calculate the n_i's, that would take $O(M)$ steps, which is impractical for large values of M. Instead, we rely on a sample of $m < M$ keys with repetitions (which we can think of as extracted from K according with the unknown probability distribution). Based on this sample, we can use the frequency of a key k, $\hat{p}_k$, as an unbiased estimator of its probability, as $E[\hat{p}_k] = p_k$. However, it is more difficult to estimate n_i with the number of keys in the sample mapped to index i, $\hat{n}_i$. If we denote by $\hat{n}_i(m)$ the number of keys out of a sample of size m mapped to index i, it is easy to verify that $E[\hat{n}_i(m)] \leq E[\hat{n}_i(m+1)] \leq n_i$. Nevertheless, one can assume that the missing contributions to $\hat{n}_i$ are from quite unprobable keys, provided a reasonable m is taken, whereby one could conclude that

$$\hat{z}_{\mathcal{H}} = \frac{1}{m} \sum_{i=1}^{N} h_i \hat{n}_i (\hat{n}_i - 1) \tag{3}$$

would provide a reasonable approximation of $z_{\mathcal{H}}$, where h_i is the number of keys in the sample that hit index i, $\sum_{i=1}^{N} h_i = m$. Notice that, by analogy with Equation 1, the frequency $\hat{q}_i$ of index i given a sample of m keys is $\hat{q}_i = \frac{h_i}{m} = \sum_{k:\mathcal{H}(k)=i} \hat{p}_k$.

Now we denote by $z(\mathcal{H}, \mathbf{p})$ the value $z_{\mathcal{H}}$ takes up when keys are distributed according to $\mathbf{p} = (p_1, p_2, \ldots, p_M)$. Similarly, we denote by $\hat{z}_m(\mathcal{H}, \hat{\mathbf{p}})$ the approximation to $z_{\mathcal{H}}$ defined in Equation 3 based on a sample of m keys with frequencies $\hat{\mathbf{p}} = (\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_M)$. We want to verify that $\hat{z}_m(\mathcal{H}, \hat{\mathbf{p}})$ converges to $z(\mathcal{H}, \mathbf{p})$ as m goes to infinity.

This allows us to write (the easy proof is omitted for the sake of conciseness):

$$\lim_{m \to \infty} \left(\min_{\mathcal{H}} \hat{z}_m(\mathcal{H}, \hat{\mathbf{p}}) \right) = \min_{\mathcal{H}} \left(\lim_{m \to \infty} \hat{z}_m(\mathcal{H}, \hat{\mathbf{p}}) \right) = \min_{\mathcal{H}} z(\mathcal{H}, \mathbf{p}). \tag{4}$$

2.1 Dynamic Key Distribution

Suppose that probability distribution $\mathbf{p}$ changes over time. We are interested in investigating how the objective function changes when $\mathbf{p}$ changes to $\mathbf{p} + \Delta\mathbf{p}$, where, of course, $\Delta\mathbf{p}$ is such that $\sum_{k=1}^{M} \Delta p_k = 0$ because $\mathbf{p} + \Delta\mathbf{p}$ is still a probability distribution. After defining $\Delta\mathbf{q}$ as the change of distribution $\mathbf{q}$ induced by a change $\Delta\mathbf{p}$ in the distribution of keys, and observing from Equation 1 that

$$\Delta q_i = \sum_{k:\mathcal{H}(k)=i} \Delta p_k, \tag{5}$$

we can write

$$z(\mathcal{H}, \mathbf{p} + \Delta\mathbf{p}) = \sum_{i=1}^{N} (q_i + \Delta q_i) n_i (n_i - 1) = z(\mathcal{H}, \mathbf{p}) + \sum_{i=1}^{N} \Delta q_i n_i (n_i - 1). \tag{6}$$

If we define $\Delta z(\mathcal{H}, \mathbf{p}) \equiv \sum_{i=1}^{N} n_i (n_i - 1) \sum_{k:\mathcal{H}(k)=i} \Delta p_k$, Equation 6 can be rewritten as

$$z(\mathcal{H}, \mathbf{p} + \Delta\mathbf{p}) = z(\mathcal{H}, \mathbf{p}) + \Delta z(\mathcal{H}, \mathbf{p}). \tag{7}$$

Given a statistical model of $\Delta\mathbf{p}$, i.e., of the dynamics of key distribution, it can be shown that, on average, this variation is quite small, so that a hash function $\mathcal{H}$ that is optimal for $\mathbf{p}$ has a high probability of being optimal for $\mathbf{p} + \Delta\mathbf{p}$ as well. Therefore, the evolutionary algorithm will be able to track the optimal hash circuit provided the key distribution does not change too fast.

2.2 A Locality-Based Model for Key Distribution Dynamics

In many applications, the distribution over K of Equation 1 will be time-dependent, in a way depending on the stochastic properties of the source. We shall now investigate how the evolutionary process adapting our circuit can be used to track this moving probability distribution in real time. Suppose keys to be addresses generated by a CPU to access memory locations while executing a program. In this case, the source will exhibit a behavior according to the well-known *locality principle* [2]. Specifically, we can expect key probability to be initially distributed according to an exponential law, with a few very probable keys and many little probable keys. Moreover, the number of keys with probability greater than a given threshold increases exponentially as the threshold goes to zero. Accordingly, we generate the initial probability distribution as follows, given a real parameter $0 < u < 1$ governing the *uniformity* of the distribution:

1. $P \leftarrow 1$;
2. $p_k \leftarrow 0$, for all $k \in K$;
3. $k \leftarrow \text{RANDOM}(1, \ldots, M)$;
4. $p_k \leftarrow p_k + (1 - u)P$;
5. $P \leftarrow uP$;
6. if $P \geq \epsilon$, go to 3, otherwise stop.

In Step 6, ϵ is the smallest positive floating-point number that can be represented in double precision.

In addition, we need to provide a law according to which the key distribution changes over time. Again, we model the scope of the locality principle allowing the probability associated to each key to change in two ways:

- by dropping to zero, when the key exits the scope of the locality, i.e. the program execution flow moves away from the address. In this case, another key gets an increase in probability of the same amount
- by averaging its value with the probability of another key. Taking the average models three different situations. First of all, it applies to a key entering the scope of the locality. This happens when one of the involved keys has zero probability, while the other has a non-zero probability value. Secondly, when both involved keys have non-zero probability, taking the average models the fluctuations in probability inside the locality scope. Finally, when both keys have zero probability, it models a null move.

Summarizing the above discussion, we describe an *atomic unit* of probability change as follows:

1. $j, k \leftarrow \text{RANDOM}(1, \ldots, M)$, such that $j \neq k$;
2. flip a fair coin:
3. if *head*, $p_j \leftarrow p_j + p_k$ and $p_k \leftarrow 0$;
4. if *tail*, $p_j, p_k \leftarrow \frac{p_j + p_k}{2}$.

We can adjust the speed of distribution change by specifying how many of these atomic changes happen per unit of time. In fact, the probability change described above leads to a uniform probability distribution in the long run. However, this does not affect our experiments which do no not last so long as to be affected by this long term behavior.

3 Evolutionary Algorithm

A population of hash circuits evolves tracking the changing key distribution thanks to an evolutionary algorithm whose features are essentially the same described in [3–5]. That is a simple generational evolutionary algorithm using fitness-proportionate selection with elitism and linear fitness scaling.

3.1 Encoding and Interconnection Map Evolution

The encoding adopted is an extension of the one described in detail in [5], allowing for the co-evolution of the interconnection map together with the circuit. For each cell, the inputs to the cell are encoded in four 16-bit integers, modulo a 64-bit mask of permissible input cells; the boolean function computed by the cell is encoded as its 16-bit truth table. Altogether, a single cell is encoded by a block of nine 16-bit integers and 64 such blocks represent a complete hash circuit.

The fitness of a circuit strongly depends on the neighborhood graph [5]. In this work, circuits are evolved without strict *a-priori* conditions on the neighborhood of cells. A condition of maximum distance has been imposed, to avoid long connections which would lead to non-routable circuits. Fig. 2 illustrates the circuit made up of 64 configurable logic blocks (CLB's). Four possible conditions are imposed on the interconnections: inputs of cells F, G, R, and S may be fed with signals coming from their neighbouring cells (marked with an "O").

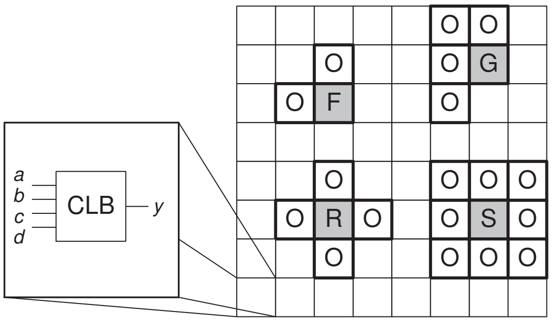

Fig. 2. Four possible boundaries imposed to a cell neighbourhood

The distance d between cells C_{ij} and C_{kl} is defined through a norm. As an example, for cell S in Fig. 2, the norm is: $d(C_{ij}, C_{kl}) = \max(|i-k|, |j-l|)$ Let us denote the maximum allowed distance with d_{max}.

Each cell in the circuit has a set of neighbouring cells (*"neighbourhood"*) $\mathcal{M}_{ij}$: $\mathcal{M}_{ij} = \{C_{kl} \mid C_{kl} \neq C_{ij};\ d(C_{ij}, C_{kl}) \leq d_{max}\}$ Possible inputs of cell C_{ij} are the external lines and the outputs of cell belonging to the set $\mathcal{M}_{ij}$.

3.2 Repair Algorithm

The random choice of interconnections may lead to unacceptable solutions. Circuits with loops are not allowed, since they would oscillate or have memory effects. A combinational circuit $\mathcal{C}$ (without loops) is always made up of an ordered set of blocks, i. e. blocks fed with input lines do precede output blocks. The ordering rule is: $C_{ij} < C_{kl}$ (C_{ij} *before* C_{kl}) if the output of cell C_{ij} is fed (either directly or through other cells) to C_{kl}. On the other hand, $C_{ij} > C_{kl}$ (C_{ij} *after* C_{kl}) if the output of cell C_{kl} is fed (either directly or through other cells) to C_{ij}. Two cells C_{ij} and C_{kl} are not related if no signal path includes both of them (partial ordering). All signal paths from input lines to the output must occur through an ordered subset of the circuit cells. Let us denote the set of cells feeding the inputs of cell C_{ij} with $\mathcal{N}_{ij}$. Of course, $\mathcal{N}_{ij} \subseteq \mathcal{M}_{ij}$. Assuming that sets $\mathcal{N}_{ij}$ have been randomly generated by the genetic algorithm, loops may occur and cell ordering is not guranteed. Therefore, it is mandatory either to *repair* circuits contining loops or to discard them.

Repair can be accomplished as follows: after a first attempt at cell ordering, the circuit cells are divided into two subsets: the *ordered* one ($\mathcal{O}$) and the *unordered* one ($\mathcal{U}$), with $\mathcal{O} \cup \mathcal{U} = \mathcal{C}$ and $\mathcal{O} \cap \mathcal{U} = \emptyset$. If $\mathcal{U} = \emptyset$, then all cells have been ordered and there is no loop. Otherwise, a loop exists in the subset $\mathcal{U}$. A cell $C_{ij} \in \mathcal{U}$ is randomly selected, and one of the cells $C_{kl} \in \mathcal{N}_{ij}$ feeding the inputs of C_{ij} is removed from $\mathcal{M}_{ij}$ (and from $\mathcal{N}_{ij}$, of course). Ordering and repair are iterated until all cells of the circuit are in the ordered set $\mathcal{O}$.

3.3 Mutation and Recombination

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. Of course, mutation can produce loops in the circuit, for some cell interconnection schemes. If this is the case, loops are subsequently removed by the repair algorithm described above. As in [5], the mutation rate is dynamically tuned while the algorithm is running to maintain a certain level of population diversity. For recombination, each cell is treated as an atomic unit of the genotype, and 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 Fitness and Selection

Minimising the objective in Equation 3 is equivalent to maximising a fitness defined as:

$$f(\mathcal{H}) = \frac{1}{1 + \hat{z}_{\mathcal{H}}} \tag{8}$$

As it can be readily noticed, such fitness is a random variable, depending on the key sample available when it is calculated. Therefore, the fitness of all individuals is recomputed at each generation, even for those that did not undergo any changes (this is always the case for the best individual). Thus, even though noisy, fitness is always consistent at each generation, being computed on the most recent key sample for all individuals. On average, an unfavourable sample will penalise the fitness of all individuals in the same way, and no bias is introduced into the evolutionary process by fitness noise. Selection probability is proportional to fitness, which is linearly scaled. Elitism is enforced throughout the algorithm.

4 Experiments and Results

In this Section we outline our experimental plan and report our main results. First of all, we observe that, in order to choose the experimentation parameters we relied on a number of previous results. Specifically, in [4] we discussed the feasibility of on-line evolution for this kind of circuits, showing that a small population size can be used for time-bounded computations without compromising the performance of our evolutionary engine. According to these results, we used for the current experimentation a population size of 16 individuals. At each generation, 4096 keys are extracted randomly, according to the current probability distribution, and the performance of each individual is assessed with respect to this sample using the genetic algorithm of Section 3. The size of the sample was chosen to be realistic with respect to implementation in hardware by means of the preprocessor module in Fig. 1. The value of u was set at $\frac{3}{4}$ in order to model a realistically restricted locality scope.

We run our algorithm for different probability distribution dynamics, each characterized by the number of atomic units of key distribution change N_C from one generation to the next. Table 1 reports our results for seven different interconnection schemes: the four schemes illustrated in Fig. 2 (F_1, G_1, R_1, and S_1) and some of their extensions to a longer distance (F_2, G_2, and R_2). Each entry in the table contains the average number of generations at which no collisions occur and the fitness value stably reaches the optimum ($f = 1$), along with the standard deviation. In one case evolution did not converge all the times.

Table 1. Experimental Results

N_C	128		256		512		1024	
	mean	stdev	mean	stdev	mean	stdev	mean	stdev
F_1	9.3	5.7	35.0	34.5	80.0	67.1	38.0	74.8
F_2	8.0	6.2	13.6	14.2	15.0	12.2	26.7	16.5
G_1	3.7	2.7	10.3	5.7	11.4	8.9	22.5	21.2
G_2	7.8	7.0	11.3	9.8	12.3	10.2	15.2	12.5
R_1	15.0	6.8	23.4	32.6	26.0	30.4	35.0	102.3
R_2	15.2	7.3	25.2	28.8	28.9	29.2	n/a	n/a
S_1	5.3	4.9	11.2	8.0	13.2	10.3	24.4	18.2

5 Sample Applications

As an example, we shall describe how a probabilistic hash function module complying with the architecture outlined in Section 1 can be used in order to adaptively associate tags to frame blocks in a *cache memory*.

In microcomputer systems, cache memories are standard, fast access components used to store copies of main memory locations often accessed by the CPU, including their addresses. Each time the CPU generates a memory address, *fully associative* cache memories use the address as an *associative tag*, simultaneously comparing it to all stored addresses (also called *tags*) in the cache memory. This comparison identifies in a single step the "right" cache location, which contains a copy of the content of the corresponding main memory location.

As the size of cache memories increased, it turned out fully-associative cache architectures to be limited by the topological complexity of the combinational network used to perform simultaneous comparisons. In *set-associative* cache memories, a set of *block identifiers* is associated via a combinational network to a smaller set of *tags*. As before, identifiers are (prefixes of) memory addresses provided by the CPU in the framework of a memory `Read/Write` operation. This time, however, tags do not correspond to single cache locations but to *sets* of locations, i.e. to cache *blocks*. For instance, in a 16-bit linear memory setting, 16-bit identifiers are mapped to a smaller set of 8-bit tags. The cache memory comprises 256 blocks, each corresponding to a tag. Each block entry contains a 8 bit prefix and 16 bits of data, so that blocks hold 512 bytes of data each, giving a total cache capacity of 64 Kbytes. When an identifier (i.e. a 16-bit address) is presented by the CPU at the cache input, a combinational network computes the tag associated to it, identifying the cache block where the value resides. Usually, this computation is performed by taking the most significant byte of the identifier as a block tag. Once the block has been found, a location is accessed inside it, using the least significant part of the identifier to find a match for the location content's prefix. This comparison can be made without excessive performance degradation [6]. In our example, since each cache block contains $l = 256$ entries, accessing the right location inside the block requires an average of $\frac{l}{2} = 128$ comparisons.

Our hash circuit supports a different technique of identifiers-to-tags association, more suitable to *adaptive cache memories*. In an adaptive cache memory, smaller blocks contain locations having high probability of being accessed, according to the probability distribution of identifiers in the input data stream. Since highly probable identifiers correspond to smaller blocks, the number of comparisons required to identify a cache location inside a block is usually shorter than in conventional cache memories. The average number of comparisons to be performed to find a location inside a block is given by

$$\frac{1}{2N}\sum_{i=1}^{N} q_i l_i$$

where N is the number of blocks, q_i is the probability of a key to be stored in the i-th block, and l_i is the length of the i-th block. This formula gives a maximum of $\frac{l}{2}$ when all blocks have an equal length l and $q_i = \frac{1}{N}$ for $i = 1, 2, \ldots, N$. As we have seen, the locality principle tells us that the probability distribution of addresses generated by the CPU changes over time. Therefore, the evolutionary engine must provide the configuration which is the current best for handling the input stream, while evolving other configurations to tolerate change.

6 Conclusion

This paper has presented a novel strategy for dynamic optimization of a hash function circuit. Experimental results show that a small population evolutionary algorithm, which could be in principle implemented in silicon, is able to effectively track a changing key distribution in real time. We intend to compare our experimental approach to available theoretical results about dynamically changing fitness landscapes [7] in order to better justify our choices in parameter settings. Comparison with other techniques for on-line circuit evolutions is also planned, as soon as suitable experimental data will be available.

References

1. Levi, D., Guccione, S. A.: GeneticFPGA: Evolving stable circuits on mainstream FPGA devices. In Proc. First NASA/DoD Workshop on Evolvable Hardware, Pasadena, CA, USA, July 1999. pp. 12–17
2. Patterson, D. A., Hennessy, J. L.: Computer Organization & Design: the Hardware/Software Interface. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1994
3. Damiani, E., Liberali, V., Tettamanzi, A. G. B.: Evolutionary design of hashing function circuits using an FPGA. In Sipper, M., Mange, D., Pérez-Uribe, A. (Eds.), Proc. Second International Conference on Evolvable Systems (ICES '98), Lausanne, Switzerland, Sept. 1998. pp. 36–46
4. Damiani, E., Tettamanzi, A. G. B., Liberali, V.: On-line evolution of FPGA-based circuits: A case study on hash functions. In Proc. First NASA/DoD Workshop on Evolvable Hardware, Pasadena, CA, USA, July 1999. pp. 26–33
5. Damiani, E., Liberali, V., Tettamanzi, A. G. B.: FPGA-based hash circuit synthesis with evolutionary algorithms. IEICE Transactions on Fundamentals, Sept. 1999
6. Burkardt, W.: Locality aspects and cache memory utility in microcomputers. Euromicro Journal, vol. 26, 1989
7. Wilke, C., Altmeyer, S., Martinetz, T.: Large-scale evolution and extinction in a hierarchically structured environment. In Adami, C., Belew, R. K., Kitano, H., Taylor, C. (Eds.), Proceedings of the 6th International Conference on Artificial Life (ALIFE-98), MIT Press, Cambridge, MA, USA, June 27–29 1998. pp. 266–274