Evolution of Non-Cryptographic Hash Function Pairs for FPGA-Based Network Applications

Roland Dobai
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
E-mail: dobai@fit.vutbr.cz

Jan Korenek
CESNET, z. s. p. o.
Zikova 4, 160 00 Prague
Czech Republic
E-mail: korenek@cesnet.cz

***Abstract*—High-speed computer networks require rapid packet processing and flexibility which can be ensured by implementing network applications in field programmable gate arrays (FPGAs). Many network applications are based on fast lookup in hash tables. It is important to use such hash functions for these tables which utilize efficiently the limited memory resources of FPGAs. Cuckoo hashing improves this utilization by using more hash functions simultaneously. However, there is no known approach for selecting those functions which together produce the best results. Bio-inspired methods are used in this paper for evolving hash function pairs for FPGA-based network applications. The evolved hash functions are based on linear and non-linear feedback shift registers and can be efficiently implemented in FPGAs. The experiments were aimed at hashing of Internet Protocol addresses and it was shown that evolved solutions can achieve better table load factor in comparison with human-created solutions.**

I. Introduction

The continuously increasing demand for high throughput of computer networks imposes challenges for implementing network applications. Hardware-based acceleration is necessary in order to meet the required throughput but high flexibility is also necessary for addressing security issues discovered later. Therefore, networking solutions based on field programmable gate arrays (FPGAs) are becoming common [1], [2].

Many network applications are based on finding records in tables, for example the packet from a given Internet Protocol (IP) address should be dropped if the address is in the table of blacklisted addresses.

Hash tables provide lookup of the records in constant time if the table load factor is low, i.e. only a small portion of the table contains records. However, the worst case lookup is proportional to the number of records. The used hash function determines how efficient will be the hashing in terms of lookup time and table load factor. A low load factor implies that for a given number of records much more memory need to be allocated which is inefficient given the limited resources of FPGAs.

Cuckoo hashing [3] improves the basic concept of hashing by using more hash functions at the same time and guarantees worst case constant lookup time which is very advantageous for high-speed network applications [2], [4]. On the other hand, it usually provides also higher load factor and therefore, more efficient memory utilization.

Hash functions are optimized in order to work well in various scenarios independently on the inputs. The optimization criteria is usually based on the fact that they should generate each output with equal probability (i.e. with uniform distribution), and minimum change in the inputs should result in a large change in the outputs [5]. However, to the best of our knowledge, there is no known approach for developing two or more hash functions which work the best *together* when used for cuckoo hashing.

Current hash functions usually produce 32-, 64- and/or 128-bit outputs. Even 32-bit outputs are too large for hash tables in FPGAs because allocation of $2^{32} = 4$ G address space is not possible in available FPGAs. Common hash tables in the largest available FPGAs require 11–14 address bits [4]. There are several solutions how to create a smaller hash value from a 32-bit value. (1) It is possible to select the required number of bits from the 32-bit value. The quality of the reduced hash might be similar to the 32-bit one but there is still the problem of making such selection that the resulting hash function would work well together with another one for cuckoo hashing. (2) The bits can be combined by XOR folding [6], i.e. some of the bits combined together by applying logic operations of exclusive disjunction (XOR). Similarly, here are also lots of possibilities for consideration.

The main contributions of the work presented in this paper are the follows.

1) Evolutionary algorithm (EA) is used for evolving hash function pairs for FPGA-based network applications. The evolved hash functions are based on linear and non-linear feedback shift registers (LFSRs, NLFSRs) and therefore, can be efficiently implemented in FPGAs. In contrast, general purpose hash functions are usually not optimized for FPGAs.
2) There is no known approach for selecting hash functions which together produce the best results for cuckoo hashing. Moreover, selecting bits from hash values produced by conventional hash functions adds further difficulties to the design process. Evaluation of all of the possibilities is not practical due to the number of possibilities. The

978-1-4799-7560-0/15 $31.00 © 2015 IEEE
DOI 10.1109/SSCI.2015.174

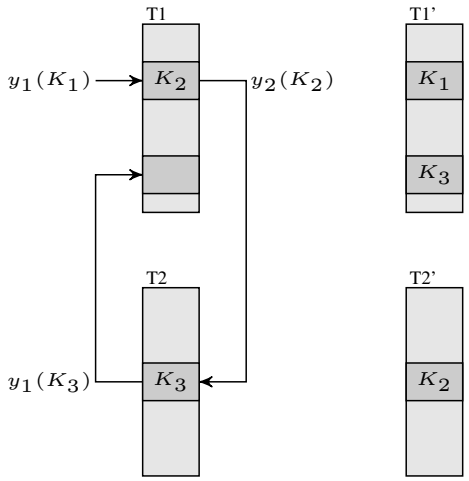

Figure 1. Principle of cuckoo hashing

proposed method finds better results than those produced by human-created solutions in reasonable time.

The rest of the paper is organized as follows. Section II describes cuckoo hashing. The related work is discussed in Section III. Section IV presents the proposed method for evolving hash function pairs for FPGA-based network applications. The proposed method is evaluated on the problem of hashing IP addresses in Section V. Section VI concludes the paper.

II. Cuckoo hashing

Cuckoo hashing [3] uses two or more tables and functions. The principle of hashing with two tables is shown in Figure 1 where T1 and T2 are tables; y_1 is the hash function of T1; y_2 the hash function of T2; and K_1, K_2 and K_3 are keys. The key is stored in a position which is directly given by the hashed value for the given table. For example, key K_2 is in position of $y_1(K_2)$ in table T1, and K_3 is in position of $y_2(K_3)$ in table T2. If a key K is searched for, then the record is checked at positions $y_1(K)$ in T1 and $y_2(K)$ in T2. This check can be done in parallel which ensures constant lookup time.

Let us assume that key K_1 needs to be inserted but the place is already taken by K_2 in T1, i.e. $y_1(K_1) = y_1(K_2)$. A common simple hash function would store both keys in the same position (bucket) which would increase the lookup time. Cuckoo hash is named after the common European bird known for brood parasitism. As a cuckoo chick pushes out eggs or youth of the host species from the nest, key K_1 pushes out K_2 from T1 and will occupy its position. Key K_2 is then moved into the other table. The position for K_2 is taken by K_3 because $y_2(K_2) = y_2(K_3)$, therefore K_2 pushes out K_3 from T2. K_3 is then inserted into T1 at position $y_1(K_3)$. The process of insertion ends because an empty place is found and no further push-outs are necessary. The final contents of tables T1 and T2 is shown as T1' and T2' in the figure, respectively. All of the keys were inserted and are accessible in constant time.

It is possible that during the insertion of K_1 this same key would be pushed-out later. In this case a *collision* exists, i.e. the key cannot be inserted. Collisions can be resolved by changing functions y_1, y_2 and rehashing all keys in T1 and T2; and/or increasing the capacity of T1 and T2. The goal of the work in this paper is to generate these new functions in order that K_1 could be inserted into the table in case of a collision arose with the previous functions.

III. Related Work

Bio-inspired methods work well in domains where the given search problem cannot be well specified [7]. For example, one can specify some characteristics (e.g. collision rate, table load factor, output uniform distribution) based on which a hash function can be considered good. However, it is not possible to directly define in advance the Boolean function which will be a good hash function. In such cases, bio-inspired methods can produce better solutions than the best human-created ones [7].

Only non-cryptographic hash functions are considered in this paper because they are simpler, faster for hash tables implemented in FPGAs. Cryptographic hash functions have other requirements mostly enforcing higher security standards [5].

Genetic programming (GP) [5], [8] and grammatical evolution [9] were used successfully for evolving hash functions. None of these can be effectively implemented in FPGAs because each candidate solution can have a different structure [5], [9] or use operations such as modulo [8], and it would be difficult to implement them in hardware. Cartesian genetic programming (CGP) [10] which reflects well the structure of FPGAs is more suitable for this purpose [11], however the chromosome is much larger in comparison with the work presented in this paper. A larger chromosome means a larger problem-space for searching, i.e. the search could be slower. In this paper LFSR and NLFSR are used which process the input sequentially: one bit at a time. This means more steps while the hash is computed but the circuit for hashing is more compacted in comparison with CGP, therefore the encoding results in shorter chromosome (genotype) and smaller problem-space. It should be noted that these steps can be performed at high frequency in FPGA because LFSRs and NLFSRs are relatively simple and can be implemented in FPGA efficiently.

The optimization for hash function development is usually done in order to maximize the avalanche effect [5], i.e. small change in the inputs should result in large change in the outputs, or minimize the collision rate [9].

To our best of knowledge, none of the existing approaches consider evolution of hash function *pairs* which would work well together for cuckoo hashing.

Furthermore, the goal of all of the previous methods is evolving universal hash functions which would work well for various scenarios. The method presented in this paper evolves new hash functions in order to resolve collisions. Therefore, the evolved functions are optimized for the given set of keys and are not intended to be universal.

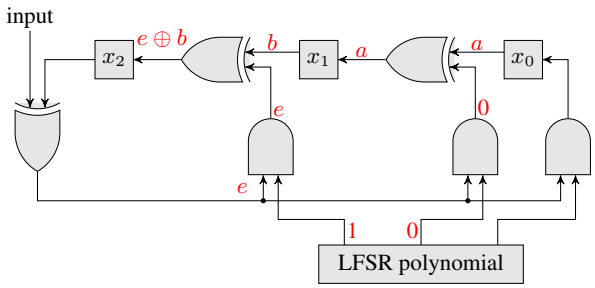

Figure 2. Proposed Galois-type reconfigurable LFSR

IV. Hash Function Pairs for FPGA-Based Network Applications

EA is employed for evolving hash function pairs where the population of candidate solutions consists of the parent and its offspring. A candidate solution is a hash function pair and is represented as genotype. Each candidate solution is evaluated and its fitness is determined. The fitness reflects how good is the candidate solution for solving the problem. The genotype and fitness function are described later in this paper.

The solution with the best fitness is selected. The other solutions are discarded and a new generation of candidate solutions is created based on the selected surviving solution. The solution kept is copied and mutation is applied. A mutation means a random change of a randomly selected part of the genotype.

The new generation is evaluated, i.e. the fitness is determined for all of the candidate solutions. This process is repeated until a desired number of generations is created and evaluated. The best solution from the last generation is the result of the search.

A. Fitness

The keys are stored in the hash table until a collision arises. The set of keys and the order in which they are processed are always the same. The number of keys successfully inserted is used as the fitness for the given candidate solution. A smaller fitness indicates a candidate solution which is worse than a candidate solution with a higher fitness.

B. Hash function pair implemented by LFSR

A candidate solution is a hash function pair encoded as two LFSR polynomials, i.e. the genotype is two LFSR polynomials. Each LFSR polynomial uploaded into the Galois-type LFSR shown in Figure 2 implements a LFSR where (x_2, x_1, x_0) is the state stored in D-type flip-flops (FFs). Galois-type LFSR contains logic gates between the D-FFs. This type was used in order to have a more balanced distribution of the logic which results in a higher maximal operational frequency in FPGA.

The feedback value e is created by merging the output of the most significant state bit x_2 with the input bit using operation XOR. The LFSR polynomial configures the LFSR as follows. A logic 0 turns off the feedback value e for the given FF stage

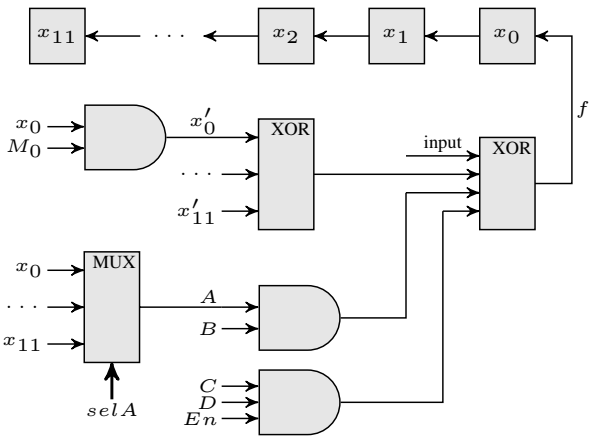

Figure 3. Proposed Fibonacci-type reconfigurable NLFSR

because $e \wedge 0 = 0$. For example, value a propagates from x_0 to x_1 because $a \oplus (e \wedge 0) = a$. A logic 1 turns on the feedback value for the stage. For example, value b gets from x_1 to x_2 mixed with the feedback value e as $e \oplus b$.

A hash computation is performed as follows. The size of the LFSR is selected based on the required number of bits of the hash value. The two LFSR polynomials configure the hash functions. One of the LFSR is initialized to all zero state and the other LFSR to all zero except x_0 which is set to logic one. This is done in order to have different seeds for the functions. The seeds does not influence the results, therefore are left unchanged during the evolution. Next, the input is applied to the LFSR. A bit is processed in each clock cycle. The hash value is the state of LFSR after all the input bits were sent into the LFSR.

A mutation means one bit flipping of randomly selected position of the LFSR polynomial, i.e. one feedback setup is flipped.

C. Hash function pair implemented by NLFSR

NLFSRs were also evaluated in an attempt to improve the results produced by LFSRs. Various other operations were enabled between the FF stages. However, the space for searching became so big and none of the runs produced better results. Several limitations were introduced in order to limit the problem-space. Firstly, Fibonacci-type NLFSR was adopted which has only one feedback function and no logic gates between the FF stages. This reduced the size of the genotype and consequently the size of the problem also. Secondly, the structure was modified for allowing one or two logic conjunctions between two NLFSR state bits. The decision was made on a list of maximum period NLFSRs [12] which have similar structure.

The developed configurable NLFSR is shown in Figure 3 where $x_0, \ldots, x_{11}$ are state bits, M_i enables/disables state bit x_i in the feedback function f for all $i \in \{0, \ldots, 11\}$; $selA$, $selB$, $selC$, $selD$ selects by a multiplexer MUX one state bit

for A, B, C, D, respectively; and En turns on/off the second, optional logic conjunction in the feedback function.

The genotype consists of two parts for the two hash functions. One part contains the following items: $(M_0, \ldots, M_{11}, selA, selB, selC, selD, En)$ where $selA$, $selB$, $selC$, $selD$ can be $0, \ldots, 11$ and the other items logic one or zero. A mutation of the candidate solution means a random change of one of these items.

The hash computation is performed similarly to the LFSR but after the last bit is sent into the Fibonacci-type NLFSR logic zeros are also sent-in until the last bit propagates trough all the state bits.

V. Experimental results

The developed framework for evolution of hash function pairs was implemented in C programming language. The experiments were run on an Intel Xeon E5-2630 processor as a single-threaded application.

A set of IP addresses from the firewall in CESNET network (Czech national research and education network) was selected and the experiments were aimed at finding a hash function pair which is able to store most of these addresses in a 8 k table. This experimental setup was inspired by real-life situations where a collision arose and subsequently new hash functions are required for rehashing the tables without collisions.

The 8 k table was divided into two 4 k tables and cuckoo hashing employed. Therefore, the required two 12-bit hashing functions were generated by 12-bit LFSRs and NLFSRs. Each run consisted of the generation and evaluation of 80 000 candidate solutions (hash function pairs). Each candidate solution was evaluated for measuring the number of records stored in the tables without collision and this number was used as the fitness of the solution.

The measurements were repeated several times and statistically evaluated based on 30 independent runs.

A. Parameter setup for the evolution

Firstly, an optimal value of the population λ was determined for the used EA. The results considering constant number of candidate solutions are shown in Figure 4. One can see that the median value improves toward $\lambda = 4$ and the lower and upper quartiles have similar tendency. The median in case of $\lambda = 8$ is the highest but the variance is the worst. Therefore, EA with 1 + 4 candidate solutions per generation was adopted (1 parent and 4 offspring).

Figure 5 contains the achieved results for various number of mutations. A low mutation rate results in lower values. Mutation rate between 6 and 10 seems to be the optimal with highest median values and lowest variations.

A similar tendency can be seen in Figure 6 for NLFSR-type of hashing and it can be concluded that 7–9 mutations produce the best results.

B. Comparison of the evolved solution with the best human-created solutions

Table I compares the evolved solutions with human-created

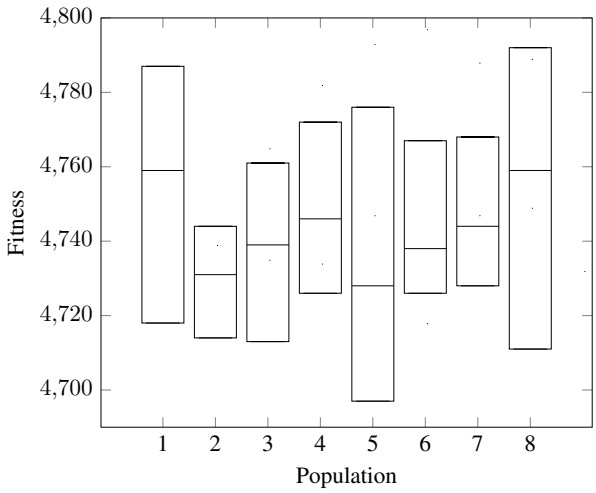

Figure 4. Influence of the population size — median, lower and upper quartile computed based on 30 independent runs

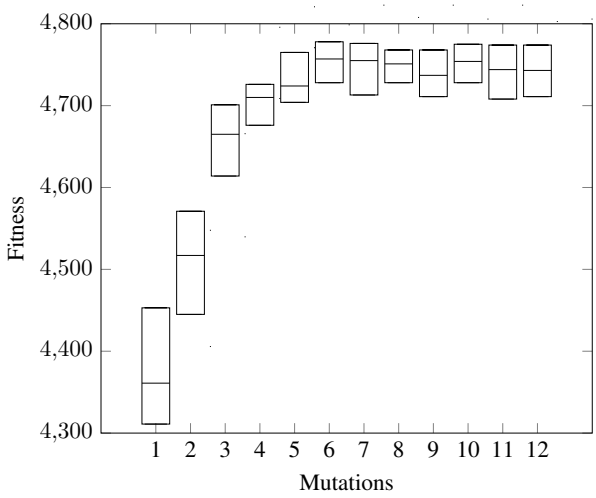

Figure 5. Influence of mutations — median, lower and upper quartile computed based on 30 independent runs

solutions. Although cyclic redundancy check (CRC) is not a hash function but it is related to LFSRs and is often used as hash function in hardware implementations [4]. Therefore, CRC32 was also included in the comparison. Only those hash functions were used which generate 32-bit hash values and can be seeded, so a pair of hash functions could be used with different seeds. The 12 least significant bits were selected from the 32-bit hash value and used for addressing. The results for 32-bit hash values reduced into 12-bit value using XOR folding [6] are marked in the table as "fold". The median of runs is marked as "med.", the lower quartile as "25%" and the upper quartile as "75%".

Column marked as "set 1" contains the results with the IP addresses used as training data during the evolution. It can be concluded that the evolved hash function pairs allow more

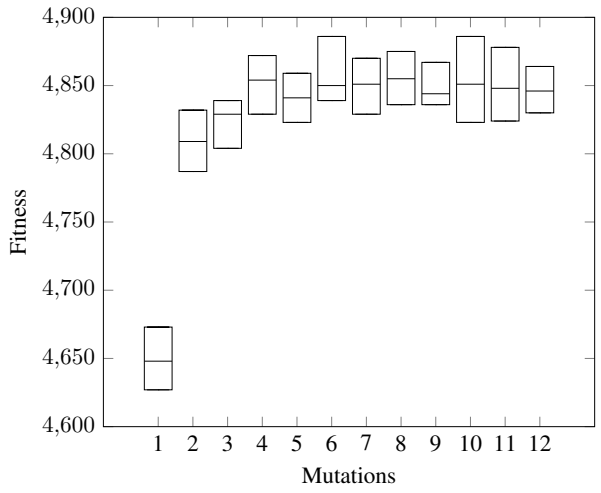

Figure 6. Influence of mutations for NLFSR — median, lower and upper quartile computed based on 30 independent runs

Table I
COMPARISON WITH COMMON FUNCTIONS USED FOR HASHING

	Set 1		Set 2	
Function	Inserted	Util. [%]	Inserted	Util. [%]
evolved NLFSR (med.)	4842	59.11	4697	57.34
evolved NLFSR (25%)	4816	58.79	3435	41.93
evolved NLFSR (75%)	4866	59.4	3498	42.7
evolved LFSR (med.)	4748	57.96	4378	53.44
evolved LFSR (25%)	4726	57.69	3838	46.85
evolved LFSR (75%)	4772	58.25	2840	34.67
CRC32	3674	44.85	3425	41.81
MurmurHash3	4199	51.26	3827	46.72
MurmurHash3 (fold)	3365	41.08	4364	53.27
SpookyHashV2	3528	43.07	3449	42.1
SpookyHashV2 (fold)	3759	45.89	4260	52
lookup3	4524	55.22	4047	49.4
lookup3 (fold)	4197	51.23	3718	45.39
fnv-1a	3787	46.23	2926	35.72
fnv-1a (fold)	3223	39.34	3557	43.42

insertions into the table and therefore, better table utilization.

One need to implement more conventional hash functions without the proposed method and select from them based on the achieved table utilization because it is not possible to known in advance which one will be the best for the given set of IP addresses. Hash function lookup3 achieved the best results among the conventional hash functions for the given set of IP addresses.

The advantage of the evolved solutions is that they are fine-tuned for the given IP addresses. As the results demonstrate, all of the evolved solutions produce better results than lookup3. The differences are approximately 200–300 more records in the table without collision. This would allow the insertion of more records into the hash tables without increasing the table size or replacing the FPGA device in case there are no memory

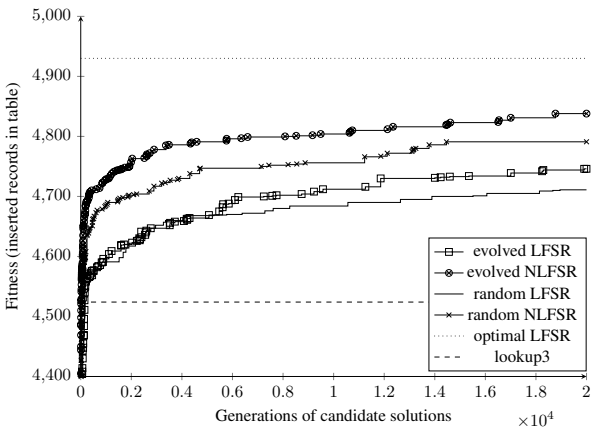

Figure 7. Comparison of inserted records in table — median values of 30 independent runs are shown

resources left.

The table evaluates the hashing functions with another set of IP addresses ("set 2"). This time the XOR-folded version of MurmurHash3 produces the best result among the conventional solutions. The performance of the same evolved solutions are worse but it is important to realize this is not the intended use of the proposed method. However, the median evolved NLFSR still achieved better results than all of the conventional hash functions. The evolution should be repeated in case the set of IP addresses changes and then the adapted hash functions should achieve better results.

The results achieved for "set 1" are also shown in Figure 7. The result of lookup3 is indicated in the figure which is the best among the conventional solutions. The best achievable (optimal) result for the proposed LFSR architecture is also available. The optimal result was achieved by evaluating all possible LFSR configurations. The computation of the optimal solution for the NLFSR architecture was not possible in reasonable computational time.

All of the searches produce very rapidly better results than lookup3. Even random search is very successful but the EA generates considerable better results.

C. Time required for evolving hash function pairs

The search for the optimal LFSR solution took 4359.17 seconds. Random search generated and evaluated 80 000 candidate solutions in 22.4 seconds and found better results than the human-crated solutions. EA required 27.1 seconds and produced even better results.

The number of possible solutions for the NLFSR architecture is much higher, therefore it was not possible to evaluate all of them and determine the optimal solution. The random search concluded the search in 93.77 seconds and the results were better in comparison with LFSR. The search based on EA in architecture NLFSR finished in 171.85 seconds.

The experiments confirmed that it is possible to evolve better hash function pairs in reasonable time in comparison

with human-created solutions. The evolved solutions become very rapidly better and can be further optimized based on the available time.

D. An example evolved solution

The best evolved NLFSR solution was able to insert 4955 IP addresses into the hash table and achieve 60.49% table utilization. This is even better than the optimal LFSR solution with 4930 insertions and can be found in 25 times shorter time. The feedback function of the first hashing NLFSR was

$$f_1 = I \oplus (x_0 \wedge x_9) \oplus (x_0 \wedge x_{10}) \oplus x_1 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_{11}$$

while the NLFSR was seeded with 0 and the second function was

$$f_2 = I \oplus (x_4 \wedge x_6) \oplus (x_7 \wedge x_8) \oplus x_2 \oplus x_6 \oplus x_9 \oplus x_{11}$$

with seed 1 where I is the input bit.

None of these two functions produces maximum period NLFSR [12] so it is very improbable that an engineer would find this solution. However, EA found out that these two functions combined together produce better results for cuckoo hashing in comparison with human-created solutions for the given set of IP addresses.

VI. Conclusions

High-speed computer networks require rapid packet processing and flexibility which can be ensured by implementing network applications in FPGAs. Many network applications are based on fast lookup in hash tables. It is important to use such hash functions for these tables which utilize efficiently the limited memory resources of FPGAs. Cuckoo hashing improves this utilization by using more hash functions simultaneously. However, there is no known approach for finding those functions which together produce the best results.

Therefore, we proposed the evolution of hash function pairs by EA for FPGA-based network applications. The evolved hash functions are based on LFSRs and NLFSRs.

The experiments were aimed at hashing of IP addresses. The EA developed hash function pairs optimized for best table load factor for the given set of addresses and size of hash table. It was shown that evolved solutions can achieve better table load factor in comparison with the best human-created non-cryptographic functions. The two 4 k hash tables were able to store by several hundred more records. Alternatively, without the proposed method the collisions could be resolved by using larger tables (i.e. two 8 k) but of course only if there would be available FPGA resources. However, the whole system need to be redesigned if the tables are replaced. The proposed method offers adaptive rehashing, therefore couple of hundred more records can be stored in the tables without redesign or replacement of the FPGA.

Currently, the experiments were performed in software-based hardware simulator only but the proposed method is intended to be implemented in FPGAs for high-speed computer network solutions because new hash functions need to be evolved rapidly in hardware. LFSRs and NLFSRs can be implemented in hardware efficiently for both high speed and low area in contrast with conventional hash functions used for comparison. Those functions are usually available for software only and hardware implementation would be less effective. The evolutionary framework for FPGAs [13] can be used for hash function development as well which would speed-up the evolution. This will be confirmed in our future work.

Acknowledgments

This work was supported by the Czech science foundation under the project Advanced Methods for Evolutionary Design of Complex Digital Circuits (14-04197S) and by the Technology Agency of the Czech Republic under the project TH01010229.

References

[1] M. Attig and G. Brebner, "400 Gb/s programmable packet parsing on a single FPGA," in *2011 Seventh ACM/IEEE Symp. Architectures for Networking and Communications Systems (ANCS)*, 2011, pp. 12–23, doi: 10.1109/ANCS.2011.12.

[2] L. Kekely, V. Pus, and J. Korenek, "Software defined monitoring of application protocols," in *2014 Proceedings IEEE INFOCOM*, 2014, pp. 1725–1733, doi: 10.1109/INFOCOM.2014.6848110.

[3] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *Algorithms - ESA 2001*, ser. Lecture Notes in Computer Science, vol. 2161, 2001, pp. 121–133, doi: 10.1007/3-540-44676-1_10.

[4] L. Kekely, M. Zadnik, J. Matousek, and J. Korenek, "Fast lookup for dynamic packet filtering in FPGA," in *17th International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, 2014, pp. 219–222, doi: 10.1109/DDECS.2014.6868793.

[5] C. Estebanez, Y. Saez, G. Recio, and P. Isasi, "Automatic design of noncryptographic hash functions using genetic programming," *Computational Intelligence*, vol. 30, no. 4, pp. 798–831, 2014, doi: 10.1002/coin.12033.

[6] "FNV hash," http://www.isthe.com/chongo/tech/comp/fnv/, [Online, accessed: 7. 2. 2015].

[7] L. Sekanina, "Evolvable hardware," in *Handbook of Natural Computing*. Springer Verlag, 2012, pp. 1657–1705, doi: 10.1007/978-3-540-92910-9.

[8] M. Safdari, "Evolving universal hash functions using genetic algorithms," in *Proc. of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, 2009, pp. 2729–2732, doi: 10.1145/1570256.1570396.

[9] P. Berarducci, D. Jordan, D. Martin, and J. Seitzer, "GEVOSH: Using grammatical evolution to generate hashing functions," in *Proc. of the Fifteenth Midwest Artificial Intelligence and Cognitive Sciences Conference*, 2004, pp. 31–39.

[10] J. F. Miller, *Cartesian Genetic Programming*. Springer Berlin Heidelberg, 2011, doi: 10.1007/978-3-642-17310-3.

[11] H. Widiger, R. Salomon, and D. Timmermann, "Packet classification with evolvable hardware hash functions - an intrinsic approach," in *2nd International Workshop on Biologically Inspired Approaches to Advanced Information Technology*, ser. Lecture Notes in Computer Science, vol. 3853, 2006, pp. 64–79, doi: 10.1007/11613022_8.

[12] E. Dubrova, "A list of maximum period NLFSRs," in *Cryptology ePrint Archive: Report 2012/166*, 2012, http://eprint.iacr.org/2012/166 [Online, accessed: 7. 1. 2015].

[13] R. Dobai and L. Sekanina, "Low-level flexible architecture with hybrid reconfiguration for evolvable hardware," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 3, 2015, art. no. 20, doi: 10.1145/2700414.