Adaptive Development of Hash Functions in FPGA-Based Network Routers

Roland Dobai, Jan Korenek and Lukas Sekanina
Brno University of Technology
Faculty of Information Technology
Brno, Czech Republic
Email: dobai@fit.vutbr.cz

***Abstract*—Accelerated network technologies are crucial for implementing packet processing in high-speed computer networks and therefore, network routers accelerated by field-programmable gate arrays (FPGAs) are becoming common. One of the time-critical jobs in routers is packet classification which requires rapid lookup in tables. Fast hash computation is a must in order to process the packets in time. Adaptive development of hash functions is proposed in this paper. The hash functions are based on non-linear feedback shift registers and configured by an evolutionary algorithm. The hash functions are developed inside of an FPGA-based network router and fine-tuned for the given table content. The experiments on the problem of hashing Internet Protocol (IP) addresses demonstrate that the evolved simple hash functions provide faster hash computation, better memory resource utilization and require smaller chip area in comparison with conventional hash functions. The best conventional hash function was able to store by a couple of hundred less IP addresses in a 8k hash table, the computation of hashes was by 42% slower, and the implementation required 15-times more hardware area.**

I. Introduction

The increasing speed of computer networks puts high requirements on packet processing. There are only a couple of nanoseconds available in future 400 Gbit/s networks for performing low-latency routing and filtering of network packets. Therefore, accelerated network technologies are crucial for implementing network applications in high-speed computer networks and solutions based on field programmable gate arrays (FPGAs) are becoming required [1], [2].

Packet filtering and classification jobs usually require finding records in tables. For example, the packet should be dropped if the Internet Protocol (IP) address of the sender is in the table of blacklisted IP addresses.

Hash tables provide a fast way of finding table records by transforming the searched key into an address by means of a hash function. It is possible that the hash function produces the same address for two or more different keys. In this case a collision exists. One of the most popular ways of resolving collisions is storing these records in linked lists [3]. This is unacceptable for FPGA-based network solutions because the lookup time of records cannot be guaranteed. The processing of packets may not be finished in time if the lookup is prolonged by sequential search in these linked list. Choosing larger hash tables can reduce the number of collisions, reduce the number of records in linked lists and consequently, reduce the lookup time. However, this approach is not practical for FPGAs because of their limited memory resources.

Cuckoo hashing [4] offers constant worst-case lookup time by allowing to be only one record at the given address in the table. It uses two or more hash functions and collisions are resolved by rehashing the record into another address by the other hash function.

Cuckoo hashing can be used for FPGA-based network routers [5] but there are some issues which require attention. Hash functions are optimized in a way that they work well for various keys. However, there is no known conventional approach for developing hash functions which work well when these functions are used *together* for Cuckoo hashing.

Probably the most important issue is the table-load factor (table utilization ratio). New hash functions should be selected and all records must be rehashed if no more records can be stored without collision in the table. Hash functions allowing higher table-load factor are desirable.

Hash functions produce usually 32-, 64- or 128-bit hash values. Even 32-bit hash values are too large for addressing in small hash tables of FPGAs. Allocation of $2^{32} = 4$ G tables is not practical in FPGAs. One can select some of the bits for addressing and discard other bits, or combine bits, for example, by XOR folding [6]. There are a lot of possibilities to be considered and there are no known conventional approaches/methods for addressing this issue.

The production of a 32-bit hash value while a part of it is discarded wastes the programmable resources of the FPGA because 32-bit operations and pipeline registers for high processing speed are expensive in terms of hardware resources. Conventional hash functions use operations like addition or multiplication in order to work well for various applications [6], [7] which might be unnecessary for hash tables with a given specific type of records. Cryptographic hash functions enforced for security are even more inefficient in term of speed of hashing and area requirements for FPGA-based hash tables, therefore they will not be considered in this paper.

Adaptive development of hash functions is proposed in this paper. The developed hash functions are based on non-linear feedback shift registers (NLFSRs) which ensures fast hash computation and requires low hardware resources in comparison with conventional hash functions. A NLFSR can

be viewed as a state automaton where the internal state is updated in each clock cycle by a one-position shift of the state and application of a non-linear function, i.e. a function consisting of AND and XOR operations.

The proposed NLFSR is fine-tuned by evolutionary algorithm (EA). The goal of the EA is to optimize the table-load factor and find such hash function solutions which work well together for Cuckoo hashing in FPGAs. The hash functions are fine-tuned also for the given type of table content.

The preliminary results were published in [8] where experiments performed on software simulator were considered only. The hardware implementation in the Zynq all programmable (AP) system-on-chip (SoC), other approaches of artificial evolution and detailed comparison including hardware-based implementation of conventional hash functions are considered only in this paper. The contributions of the work presented in this paper are as follows. (1) The developed hash functions are optimized for high table-load factor and for Cuckoo hashing. Better utilization of memory resources ensures storage and fast lookup of more table records without the need of rehashing or replacing the FPGA device with a one with more programmable resources. (2) The developed hash functions ensure fast hash computation with low requirements of hardware resources in FPGAs in comparison with human-created hash functions. Fast hash computation is crucial for FPGA-based network routers.

The rest of the paper is organized as follows. Cuckoo hashing is introduced in Section II. The related work is discussed in Section III. Section IV describes the proposed adaptive development of hash functions for FPGA-based network routers. Section V is devoted to the case study of IP address filtering. The achieved results are evaluated in Section VI and Section VII concludes the paper.

II. Cuckoo hashing

Cuckoo hashing [4] ensures that at most one key is stored in a given table location. This results in worst-case constant lookup time because it is guaranteed that there is no additional search in linked lists after the table location was determined. Potentional collisions are resolved by rehashing the table record into another table position by means of an additional hash function.

The principle of Cuckoo hashing using two hash functions is demonstrated in Figure 1 where T is the hash table; K_1, K_2, K_3 are keys; and y_1, y_2 are hash functions. Let us assume that keys K_1 and K_2 are already stored in hash table T and the insertion of key K_3 requires the replacement of table records because $y_1(K_1) = y_1(K_3)$, i.e. the key is hashed to an occupied table position. The replacement is performed as follows. (1) K_3 pushes-out K_1 from the table. (2) The pushed-out key K_1 is rehashed with the other hash function y_2 into another table location. Let us assume that this location is occupied as well, by K_2. Then K_1 pushes-out K_2 from the table. (3) K_2 is rehashed by the other hash function y_1 to another table location. This location is not occupied, therefore

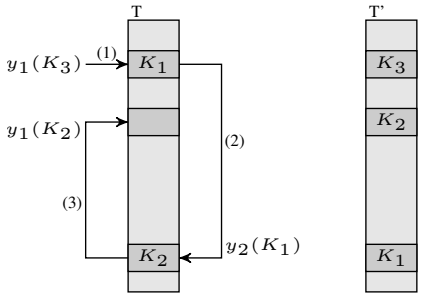

Figure 1. Record insertion with cuckoo hashing

the replacement ends. The table content after the resolution is shown in table T' in the figure.

It is possible that the same key is pushed-out again during the replacement. In this case, replacement is not enough and whole table need to be rehashed with new hash functions.

III. Related Work

Hash functions can be developed by optimizing some desired characteristics, e.g. collision rate, table-load factor, output uniform distribution [7]. This will lead to general-purpose hash functions which *probably* would work well in various scenarios. However, one cannot specify the hash function which would fit the current needs exactly. Bio-inspired methods usually work well in application domains where the search problem cannot be well defined [9].

Genetic programming (GP) [7], [10] and grammatical evolution [11] were successfully used for evolving hash functions. However, the evolved solutions cannot be implemented effectively in FPGAs because the candidate solutions have variable size and the reconfiguration of candidate solutions might be difficult. The structure of the candidate solutions contains elements which cannot be effectively implemented in hardware (e.g. modulo division) or would require a large chip area (e.g. multiplication).

Cartesian genetic programming (CGP) [12] uses a structure with bounded area which makes it advantageous for FPGA-based evolvable hardware [9]. CGP-based hash function evolution has the disadvantage that the structure of the candidate solution can be encoded by larger chromosome [13] in comparison with the work presented in this paper. Therefore, the search space is bigger, the evolution slower and it is more difficult to find suitable solutions.

The hash function development is usually guided by the avalanche effect [7], i.e. small change in the inputs should result in large change in the outputs, or minimize the collision rate [11]. These approaches would contribute to developing hash functions for general use.

The work presented in this paper is aimed at the development of hash functions which are fine-tuned for the given table content. The hash functions are optimized for table-load factor and are developed for using with Cuckoo hashing, therefore they are optimized for working well together. They use simple operations which can be effectively implemented in FPGAs.

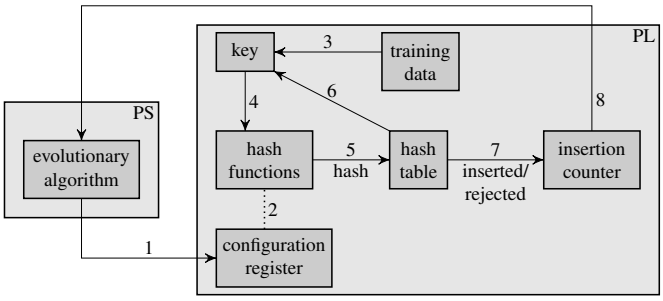

Figure 2. Overview of the system implemented in Zynq AP SoC

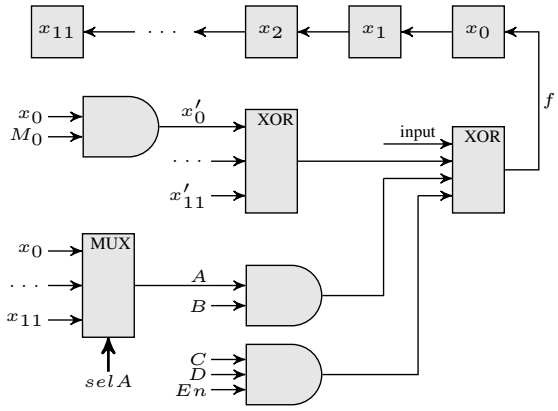

Figure 3. Proposed reconfigurable NLFSR architecture for hashing

IV. Development of hash functions in FPGA

The proposed adaptive development fine-tunes the evolved hash functions for Cuckoo hashing and also for the given table content. The development is based on the creation and evaluation of hash functions which are represented as NLFSRs. The creation is performed by the use of bio-inspired operators. The evaluation of candidate solutions is based on the insertion of the desired content into the hash table as it was described in Section II. The evaluation computes the fitness value for the solution as the number of records which can be inserted without collision into the hash table. It is assumed that a set of training records is available. For example, the hash table T' in Figure 1 contains three keys. If the insertion of the fourth key would fail then the candidate solution (hash functions y_1 and y_2) would obtain 3 as the fitness value.

The goal of the EA is finding candidate solutions with the highest fitness. This is done through the use of bio-inspired operators: selection and mutation. The search starts with random initial population of candidate solutions. The solutions are evaluated and the one with the highest fitness is selected for reproduction. This solution together with offspring solutions creates the new generation. The offspring solutions are created by mutating the best solution. The mutation operator changes h randomly selected parts of the solution. If the EA is well tuned for the particular problem then candidate solutions with higher and higher fitness can be found in the process of evolution. This means that it is possible to improve the fitness starting with random candidate solutions and eventually develop hash functions which are able to store a lot of records in the hash table.

A. The system for evolution implemented in FPGA

The system for evolution implemented in the Zynq AP SoC is shown in Figure 2. The evolution of candidate solutions is performed as follows. (1) The search is guided by the EA which is running in the ARM-processor-based processing system (PS) of Zynq. The PS is available in the Zynq architecture without any additional implementation cost. Candidate solutions are generated and are established in the programmable logic (PL) by writing the chromosome into the configuration register. The chromosome is the NLFSR configuration which represents the candidate solution (hash functions). (2) The configuration register sets the hash functions by reconfiguring it through multiplexers. (3) The key is extracted from the memory (training data). (4) The hash function computes the hash for the given key. (5) The insertion of the key is attempted into the hash table to the address given by the hash. (6) If the key pushes-out another key from the hash table then the steps from 4 are repeated and the pushed-out key is rehashed into another position. (7) The insertion ends by either successfully storing all the keys which were pushed-out as the consequence of inserting the key in step 3; or detecting an unresolvable collision, i.e. the same key is pushed-out twice. If all keys were successfully inserted then the insertion counter is incremented and the insertion continues from step 3 with another key. (8) If a collision was detected then the inserted number of keys is sent to the EA. The EA assigns this number to the candidate solution as the fitness which measures the quality of the solution. The process is repeated from step 1 until the desired number of generations are created and evaluated.

B. Proposed NLFSR-based reconfigurable architecture

The candidate solution is established in the PL by writing the chromosome into the configuration register. The chromosome contains the following items for each hash function under the assumption that 12-bit hashes are desired: $(M_0, \ldots, M_{11}, selA, selB, selC, selD, En)$ where M_i turns on/off the state variable x_i for XOR in the feedback function for all $i \in \{0, \ldots, 11\}$; $selA, \ldots, selD$ select state bits into the AND gates; and En turns on/off the second optional AND gate. The proposed configurable NLFSR for hash computation is shown in Figure 3 where the key is applied to the NLFSR sequentially bit-after-bit through "input" and the hash is stored in $x_0, \ldots, x_{11}$ after all input bits were processed.

The proposed NLFSR architecture can be set up to generate as many bits as required for addressing in the hash table. Furthermore, it contains only simple one-bit AND and XOR gates which require low chip area.

The proposed reconfigurable architecture can create NLFSRs with one or two AND gates in comparison with general NLFSRs which does not place any restriction on

the feedback function. This constrain was incorporated into the proposed architecture in order to reduce the number of possibilities which are considered during search and therefore, find acceptable solutions in shorter time. The choice was motivated by the fact that NLFSRs with maximum period used in practice contain only one or two AND gates in the feedback function [14].

Linear feedback shift registers were previously considered also but the preliminary experiments proved that NLFSRs can be more advantageous for hash computation [8].

V. Case study: IP address filtering

Let us assume that there is a given list of suspicious IP addresses and all communication originating from these addresses must be monitored in the FPGA-based network router (monitor). Then IP filter must be implemented in the monitor in order to filter-out packets from the listed IP addresses. The suspicious IP addresses are stored in a hash table using Cuckoo hashing in order to be able to determine rapidly a hit or miss for the transfered packet.

Further assume that the list of suspicious IP addresses will be extended in time because additional addresses need to be filtered. As a consequence, it is possible that the addresses cannot be inserted into the hash table even after the replacement of the hash functions and rehashing the table.

Let us consider the filtering of 32-bit IPv4 addresses and a hash table with $2^{13} = 8$ k records requiring 13-bit hashes for addressing the records, or two 12-bit hashes when Cuckoo hashing with two hash functions is used.

The discussion of three approaches follow for designing hash functions for IP address filtering: the conventional and two unconventional approaches based on EAs.

A. Conventional approach

The conventional approach for this case study would be the pre-implementation of several general-purpose hash functions.

Another hash function would be selected in the case when rehashing is required. However, it not guaranteed that rehashing will resolve the collision even when all the pre-implemented hash functions are considered. One might face the fact that no additional IP address can be inserted into the list. In this case, the redesign of the system is required: the use of a larger hash table is needed if there are free memory resources available in the FPGA, or the replacement of the FPGA device otherwise.

B. State-of-art unconventional approach based on CGP

CGP can be considered as the state-of-the-art unconventional approach for evolving solutions in FPGAs [9], [12]. It is possible to evolve hash functions with sequential or parallel hash computation. The parallel implementation is created by "unrolling" the iterative loop of the sequential implementation, therefore it requires more hardware elements [15]. More configurable hardware elements result in longer chromosome for encoding the candidate solution and consequently, the problem

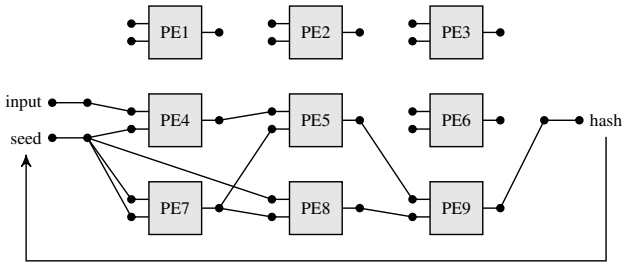

Figure 4. Array of PEs of CGP showing a candidate hash function

space becomes bigger and the search slower. Therefore, sequential hash computation is considered for the unconventional approach based on CGP and for the proposed NLFSR-based approach as well.

CGP develops solutions in a two-dimensional array of processing elements (PEs) resembling the structure of FPGAs. The reconfigurable array of 12-bit PEs is shown in Figure 4 where "input" is the 8-bit input value, "seed" is the 12-bit input seed and "hash" is the 12-bit hashed output under the assumption that 12-bit hash values are required for addressing. The PEs are able to implement multiplication, bitwise XOR and rotation operations as it was determined as the best set of elementary functions for hash function evolution (optimized for the quality of solutions) [7]. The use of the multiplication is necessary which is very unfortunate because its implementation requires considerable chip area and multiplication must be present in all of the PEs.

The IP address is hashed by the candidate solution as follows. The first byte of the address is sent into the PE array through input with seed value 0. The value propagates through the PE array while the hash is computed. This process is repeated with all the four bytes of the IP address. The hash output for the previous byte is used as the seed for the current byte. The hash value of the IP address is the hash output after all four bytes were processed.

The EA is used for finding suitable candidate solutions by changing the interconnections between PEs. The number of records insertable into the table without collision is used as the fitness value measuring the quality of candidate solutions. The evolution is started by random initial population and each population contains the candidate solution with the highest fitness from the previous generation and 4 offspring solutions created by mutations $h = 8$ from the best candidate solution. The PE array size was set to 8×4. These parameters were determined after optimization for the highest table-load factor.

One of the advantages of implementing a PE array in FPGA is that each PE column can represent a pipeline stage using registers at the PE outputs. This means that the candidate solution can be evaluated rapidly because the PE array works at high operational frequency and produces an output in each clock cycle with initial latency equal to the number of PE columns. This advantage cannot be exploited for the given case study because the hash output must be sent into the PE array for the new seed together with the next input. In other

words, the next input can be sent into the PE array only after the hash of the previous input is already available. This means that the high operational frequency must be sacrificed or the latency will delay the computation of each input. In either way the hash computation will be slower than expected.

C. Proposed unconventional approach based on NLFSR

We propose to use the adaptive development of hash functions using EA for IP address filtering in FPGA-based routers.

The hash function is developed in a reconfigurable NLFSR as shown in Figure 3. The 32-bit IP address is processed by the NLFSR bit-after-bit and sent-in through input. $12-1$ additional zeroes are sent-in after the IP address in order to propagate the last bit through all the state bits. The hash of the IP address is stored in the register $(x_0, \ldots, x_{11})$ after finishing these steps.

The training data for the EA is the IP address list. The number of records insertable into the table without collision is used as the fitness value measuring the quality of the candidate solution.

VI. Experimental results

The proposed development of hash functions for IP address filtering was implemented in an XC7Z020 Zynq AP SoC device. The IP addresses were obtained from a firewall in the national research and education network. 32-bit IPv4 addresses and a hash table with 8 k records requiring two 12-bit hashes for Cuckoo hashing were used.

The hardware simulator for the solution based on CGP was implemented in the C programming language. C/C++ implementations of conventional hash functions were adopted from [16]. The software-based experiments were run on an Intel Xeon E5-2630 processor using only one thread.

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

A. Parameter optimization for high table-load factor

The fitness for a candidate solution is the number of IP addresses which can be stored in the table without collisions. The candidate solutions are optimized for high fitness and therefore high table-load factor.

The EA evolves solutions in the proposed NLFSR architecture. A simple $(1 + \lambda)$ EA was employed which means that the population always consists of the best candidate solution from the previous generation and λ offspring solutions created by mutations of the best candidate solution.

Various values of λ were considered in the experiments and the achieved results are shown in Figure 5 where the median, lower and upper quartile fitness values are shown considering the $80\ 000$ candidate solutions were generated and evaluated in each experiment. It can be observed that the median value is the highest for $\lambda = 1$ and 8. The second largest value was achieved for $\lambda = 4$, but for considerable lower variance, therefore this value was selected as the most useful setting for λ.

The subsequent experiments were aimed at investigating the impact of the mutation-rate on the fitness value. The achieved

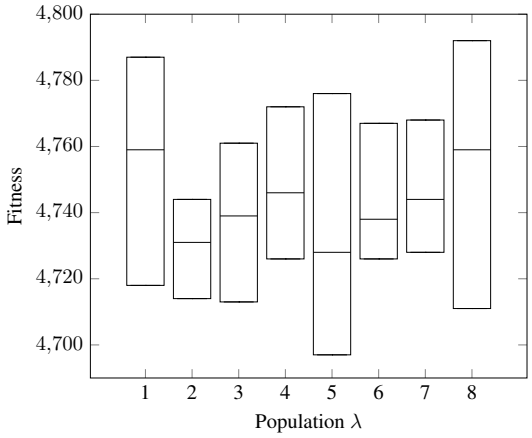

Figure 5. Fitness influenced by the population size

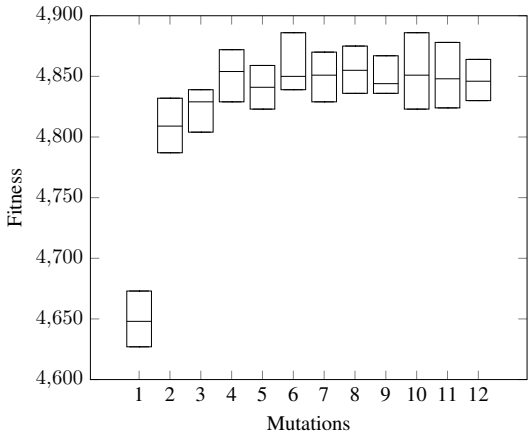

Figure 6. Fitness influenced by mutations

results for various number of mutations are shown in Figure 6. The fitness value improves while the number of mutations is increased from 1 until 8. Further increase of the mutation-rate indicates the degradation of the fitness value as well as the increase of the variance between the results from independent runs. Therefore, $h = 8$ mutations were selected which is the most useful setting for h.

B. Comparison with state-of-the-art solutions

The evolved hash functions based on NLFSR were compared with conventional and unconventional solutions. The conventional solutions include 32-bit human-created non-cryptographic hash functions which support seeding. Hash functions without seed, creating only 64-bit or larger hashes and cryptographic hash functions were not considered. Hash functions evolved by CGP were evaluated as the state-of-the-art unconventional solution for FPGA-based evolvable hardware. Table I contains the achieved results where "med" indicates the median solution, 25% the lower quartile and 75% the upper quartile solution. The median solution is considered in a fair comparison where the solution from only one run is used. Best solution can be used where there is enough time for

Table I
COMPARISON WITH COMMON FUNCTIONS USED FOR HASHING

	Set 1 (training)		Set 2		Set 3		Set 4	
Function	Inserted	Util. [%]	Inserted	Util. [%]	Inserted	Util. [%]	Inserted	Util. [%]
evolved NLFSR (med.)	4850	59.2	3567	43.54	4322	52.76	3806	46.46
evolved NLFSR (25%)	4831	58.97	3057	37.32	3508	42.82	4495	54.87
evolved NLFSR (75%)	4862	59.35	4596	56.1	4231	51.65	3443	42.03
CGP (med.)	4723	57.65	3293	40.20	3388	41.36	3882	47.39
CGP (25%)	4682	57.15	3934	48.02	3207	39.15	3725	45.47
CGP (75%)	4758	58.08	4233	51.67	3965	48.4	2780	33.94
CRC32 (crop)	3674	44.85	3425	41.81	3401	41.52	4176	50.98
MurmurHash3 (crop)	4199	51.26	3827	46.72	4179	51.01	3384	41.31
MurmurHash3 (fold)	3365	41.08	4364	53.27	3839	46.86	3074	37.52
SpookyHashV2 (crop)	3528	43.07	3449	42.1	4121	50.31	2176	26.56
SpookyHashV2 (fold)	3759	45.89	4260	52	4062	49.58	2128	25.98
lookup3 (crop)	4516	55.13	4047	49.4	3996	48.78	3951	48.23
lookup3 (fold)	4140	50.54	3656	44.63	3702	45.19	2914	35.57
fnv-1a (crop)	3787	46.23	2926	35.72	1995	24.35	2583	31.53
fnv-1a (fold)	3223	39.34	3557	43.42	2326	28.39	3949	48.21

performing several runs and selecting the best solution among all the runs. The required 12-bit hashes from the 32-bits of the conventional solutions are created either by using only the 12 least significant bits, or combining the 32 bits into 12 bits using XOR folding [6]. These variations are marked as crop and fold in the table.

The table contains the achieved fitness (inserted records) and table utilization for various IP address sets. Set 1 was used for training the proposed method based on NLFSR and CGP as well. It can be observed that the proposed evolved solutions achieve higher table utilization (table-load factor) than CGP and all of the human-created hash functions. The median evolved solution is able to achieve 59.2% table utilization while the median CGP 57.65% and lookup3, the best human-created solution, only 55.13%. Lookup3 was created by Bob Jenkins, "Oracle's resident expert on hash functions" whose hash functions are used by "Infoseek, Dreamworks (Shrek), Perl, Ruby, Linux, and Google" [17].

The fitness development of the median solutions of the proposed NLFSR and CGP are compared with lookup3 in Figure 7. It can be observed that the CGP solution overperforms lookup3 very rapidly after just a few generations. The proposed NLFSR-based architecture achieves using the trivial random search better results than both of them. NLFSR tuned by EA further improves the results.

The proposed method improves the table utilization by several hundred IP addresses which can be considered significant because the hash table can be used longer without rehashing the whole table which would mean putting the IP filter offline for considerable time.

The results for sets 2, 3, 4 in Table I demonstrate how well each hash function performs with other IP addresses unseen during the evolution. It should be noted, that the evolved solutions were trained with set 1 and re-evaluated with the other sets without repeating the evolution. This is not the intended use of the proposed method but it demonstrates that the results are acceptable for other sets as well. The developed solutions are able to insert successfully 3567, 3057 and 4596 records into the hash table using IP set 2. However, these results improve to 4731, 4690 and 4763 if the solutions are re-evolved for the given new IP set. The upper quartile evolved solution without re-evolution achieves the best result for set 2. The median solution dominates set 3 and the lower quartile evolved solution set 4.

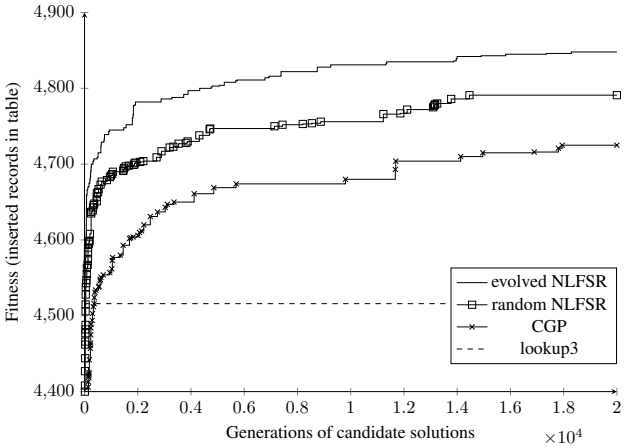

Figure 7. Comparison of inserted records in table

A different hash function achieves the best results among the conventional solutions for each four examined sets. Therefore it is not possible to select one which would always be the most successful. One should implement several of them and evaluate all of them during collision resolution and rehashing. The success however cannot be guaranteed. The proposed solution offers the advantage that the hash can be fine-tuned for the given IP set and achieve better table-load factor with

Table II
TRANSFORMATION OF THE EVOLVED SOLUTION INTO PARALLEL ARCHITECTURE

x_{11}	x_{10}	...	x_2	x_1	x_0	I
0	0	...	0	0	0	i_0
0	0	...	0	0	i_0	i_1
0	0	...	0	i_0	$i_0 \oplus i_1$	i_2
0	0	...	i_0	$i_0 \oplus i_1$	$i_0 \oplus i_1 \oplus i_2$	i_3
⋮	⋮	⋮	⋮	⋮	⋮	⋮
F_{11}	F_{10}	...	F_2	F_1	F_0	

reasonable probability in comparison with conventional hash functions.

C. Time required for evolving solutions

The main advantage of conventional hash functions is that they do not have to be tuned and therefore are available almost immediately when one does not consider the implementation time.

However, several minutes of delay required for evolution can be considered acceptable in the scenario where the IP addresses are inserted manually into the filter. The evolution consisting of the generation and evaluation of 20 000 generations of 4 candidate solutions took 87 seconds (in average) in the PL of the Zynq AP SoC. For comparison, the evolution of the CGP-based solution took 168 seconds in the hardware simulator.

The fitness improves very rapidly as it can be seen in Figure 7. Therefore it is possible to stop the evolution earlier and reduce the time of evolution. Considering just 2 000 generations reduces the evolution time to less than 9 seconds and the evolved solution will be still better than the CGP-based solution after 20 000 generations.

D. Example evolved solution

The best evolved solution achieves fitness 4972. The feedback function for the first hash is

$$f_1 = I \oplus (x_4 \wedge x_7) \oplus x_2 \oplus x_3 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_{11}$$

using initial seed 0 and for the second one is

$$f_2 = I \oplus (x_4 \wedge x_6) \oplus x_0 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_6 \oplus x_8 \oplus x_9 \oplus x_{11}$$

using initial seed 1 where $x_0, \ldots, x_{11}$ are state registers shown in Figure 3 and I is the one-bit input.

Sequential hash functions are evolved because they can be encoded by shorter chromosomes and their evolution is faster. Furthermore, the used evolutionary framework cannot fully benefit from a parallel structure because there is sequential dependence between the IP addresses as they push-out each other from the hash table. However, it is possible to transform the evolved hash functions into a parallel architecture for applications which can benefit from it. The process is demonstrated in Table II and is performed as follows. For each state register a constant Boolean function is created based on the seed value. Consequently, the input bits $i_0, \ldots, i_{31}$ are shifted-in. The Boolean function for the feedback is computed based on the current one-bit input and the given state of the register. This Boolean function is shifted into x_0 and all the state bits are shifted by one position from x_0 to the direction of x_{11}. The Boolean functions $F_0, \ldots, F_{11}$ are finished after all the input bits were processed. The implementation of these functions represents the parallel implementation of the hash function because in just one step it is able to compute the hash based on the input bits and the seed value.

E. Comparison of FPGA-implementations

The reconfigurable elements required for implementing the hash solutions are shown in Table III. Column "evolved" contains the elements required for the proposed evolvable and reconfigurable NLFSR architecture. One can reimplement the evolved solution without the possibility of further reconfiguration. These results are shown in column non-reconf. It can be concluded that the reconfigurable features require about 50% area overhead. The table contains the implementation of lookup3 which produced the highest table load-factor among the conventional solutions for IP set 1 in Table I. The implementation uses pipelining in order to work at the same operational frequency and computes the IP address byte-after-byte. The results indicate that lookup3 requires by 42% more clock cycles than the NLFSR for computing the hash for the IP address. The 50 cycles of NLFSR consist of 32 cycles for processing the 32-bit IP address, 11 cycles of shifting-in zeros for propagating the last input through all state bits and the remaining 7 clock cycles are the communication latency with the environment.

The evolved solutions produce not just higher table-load factor but also provide faster hash computation which is very important in high-speed network routers. Moreover, the implementation of lookup3 requires 15 times more reconfigurable elements because it uses 32-bit operations and the hash computation requires the application of XOR operations, rotations and subtractions, all of these 7 times, and 2 additions.

The system column contains the implementation results for the whole system which includes even the bus communication and synchronization between clock domains; and the core column the implementation of the evolutionary framework only. The implementation of lookup3 requires two times more chip area than the whole evolutionary framework for developing adaptable hash functions.

The block random-access memory (BRAM) tiles indicated in the table are used for implementing the hash tables.

VII. CONCLUSIONS

Adaptive development of hash functions in FPGA-based network routers is proposed in this paper. The developed hash functions are based on NLFSR which ensures fast hash computation and requires few hardware resources in comparison with conventional hash functions. The proposed NLFSR is fine-tuned by EA. The table-load factor is optimized and such hash functions are developed which work well together for

Table III
COMPARISON OF IMPLEMENTATIONS IN ZYNQ AP SOC

	evolved	non-reconf.		lookup3		system	core
			diff. [%]		diff. [%]		
clock cycles	50	50	0	71	42		
slice LUTs	40	19	−53	638	1495	897	240
slice registers	60	31	−48	1034	1623	1329	419
multiplexers	4	0	−100	0	−100	8	8
BRAM tiles	0	0	0	0	0	17	17

Cuckoo hashing in FPGAs. The hash functions are fine-tuned also for the given type of table content.

The proposed method was evaluated on the problem of IP address hashing. The experiments demonstrated that it is possible to evolve hash functions which are better than human-created solutions in terms of hash computation speed, memory utilization through higher table-load factor and chip area requirements.

The evolved hash functions work sequentially and are still faster than the parallel implementation of lookup3, the human-created solution which achieved the highest table-load factor among the conventional solutions (but lower than the evolved solution). The framework for developing hash functions require some additional chip area but the evolved hash functions have significantly lower chip area requirements than lookup3.

The evolution of sequential hash functions is faster and might create better solutions because the used chromosome is shorter. The hashing framework in the surveyed case study cannot fully benefit from a parallel hash function as it is suggested by the results achieved with the parallel implementation of lookup3. However, it is possible to transform the sequential solution into a parallel one. The pipelined parallel implementation would be able to produce a hash in each clock cycle with some initial latency. This will be explored in our future work.

The main contribution of the work presented in this paper is the development of hash functions which is adapted to the given type of table content. Alternatively, one could implement several conventional hash functions and switch between them when the current hash function gives inadequate results. The proposed adaptive development can produce better results that it would be possible with pre-generated conventional solutions. The proposed hash functions are able to store couple of hundred more records in the table in comparison with the best human-created hash functions. This means that the given hash function can be used longer without rehashing which would require putting offline the filtering capabilities of the router.

The proposed method was developed primarily for FPGA-based network routers but can be used in application specific integrated circuits as well because the hash functions are configured by writing into user-made configuration registers and the native reconfiguration of FPGAs is not used.

This work was supported by the Czech science foundation under the project GP16-08565S and by the Ministry of the Interior of the Czech Republic under the project VI20152019001.

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] A. G. Konheim, *Hashing in Computer Science: Fifty Years of Slicing and Dicing*. New Jersey, USA: Wiley-Interscience, 2010.

[4] 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.

[5] 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.

[6] "FNV hash," http://www.isthe.com/chongo/tech/comp/fnv/, [Online, accessed: 3. 20. 2016].

[7] 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.

[8] R. Dobai and J. Korenek, "Evolution of non-cryptographic hash function pairs for FPGA-based network applications," in *2015 IEEE Symposium Series on Computational Intelligence (International Conference on Evolvable Systems - ICES)*, 2015, pp. 1214–1219, doi: 10.1109/SSCI.2015.174.

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

[10] 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.

[11] 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.

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

[13] 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.

[14] E. Dubrova, "A list of maximum period NLFSRs," in *Cryptology ePrint Archive: Report 2012/166*, 2012, http://eprint.iacr.org/2012/166 [Online, accessed: 3. 20. 2016].

[15] S. Kilts, *Advanced FPGA Design: Architecture, Implementation, and Optimization*. New Jersey, USA: Wiley-IEEE Press, 2007.

[16] "SMHasher, a test suite designed to test the distribution, collision, and performance properties of non-cryptographic hash functions," https://github.com/aappleby/smhasher, [Online, accessed: 3. 17. 2016].

[17] "Bob Jenkins, Software Developer," http://burtleburtle.net/bob/other/resume2.html, [Online, accessed: 3. 17. 2016].