Applied Soft Computing 56 (2017) 173–181

Contents lists available at ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

Evolutionary design of hash function pairs for network filters

Roland Dobai *, Jan Korenek, Lukas Sekanina

Brno University of Technology, Faculty of Information Technology, Centre of Excellence IT4Innovations, Czech Republic

ARTICLE INFO

Article history:
Received 19 August 2016
Received in revised form 22 February 2017
Accepted 6 March 2017
Available online 12 March 2017

Keywords:
Evolutionary algorithm
Hash function
Network filter
Field-programmable gate array
Cuckoo

ABSTRACT

Network filtering is a challenging area in high-speed computer networks, mostly because lots of filtering rules are required and there is only a limited time available for matching these rules. Therefore, network filters accelerated by field-programmable gate arrays (FPGAs) are becoming common where the fast lookup of filtering rules is achieved by the use of hash tables. It is desirable to be able to fill-up these tables efficiently, i.e. to achieve a high table-load factor in order to reduce the offline time of the network filter due to rehashing and/or table replacement. A parallel reconfigurable hash function tuned by an evolutionary algorithm (EA) is proposed in this paper for Internet Protocol (IP) address filtering in FPGAs. The EA fine-tunes the reconfigurable hash function for a given set of IP addresses. The experiments demonstrate that the proposed hash function provides high-speed lookup and achieves a higher table-load factor in comparison with conventional solutions.

1. Introduction

Computer networks are a potentially dangerous environment where the integrity and the security of shared data can be violated by an attacker. The monitoring and filtering of the communication can be a countermeasure against attacks and other unlawful activities such as the illegal sharing of copyrighted data. After a monitoring network node receives a data packet the processing must be completed in the time given by the speed of the network. For example, in future 400 Gbps networks there are only a couple of nanoseconds available for performing all of the required operations on the packet. On the other hand, the processing of the packets requires time consuming operations such as finding records in tables, updating data in other tables and external memory accesses. The performance of general-purpose processors is insufficient and, therefore, network monitors and filters are implemented and accelerated, for example, in field-programmable gate arrays (FPGAs) [1,2].

A common identification of the attacker is based on its Internet Protocol (IP) address. The network monitor needs to lookup the source IP address of the packet in various tables, e.g. in a table of nodes for monitoring and blacklisting. These tables are implemented usually as hash tables with constant *worst-case* lookup time [3,4]. Hash tables with linear worst-case lookup cannot be used because it is impossible to guarantee that the packet will be processed in time. Linear lookup is the result of mapping more than one IP addresses to the same table position which means that all table records in the given position need to be compared during the lookup.

Constant worst-case lookup can be achieved by perfect hash functions [5] which map each IP address to a unique position in the table and no additional search is required after identifying the table position. However, the hash function has a relatively large memory overhead requiring at least 2 bits per each IP address [6]. Insertion of additional IP addresses into the table requires the rebuilding (rehash) of the table which can take considerably longer than the time available for filtering because the filter must be put offline for rehashing, or a switch to an alternative filter is required. The disadvantages of perfect hashing motivated the researchers to consider cuckoo hashing [7] as an alternative for hashing in FPGAs [8–11].

Cuckoo hashing uses two or more hash functions. These functions map items to a different part of the hash table. The insertion of a new item into the table is performed as follows. The hash is computed for the item which determines its position in the table. If the item is mapped to an occupied position then it pushes out the previous occupant from that position just like the offspring of the namesake European brood-parasitic bird cuckoo pushes out the other eggs from the nest. The pushed-out item is rehashed by another hash function into a different position of the hash table. Cuckoo hashing with two hash functions h and q is shown in Fig. 1 where item 1 is hashed by h into the table, and item 2 is pushed-out and is rehashed by q elsewhere into the table. The items are repeatedly pushed-out and rehashed by using the available hash

* Corresponding author.
E-mail addresses: dobai@fit.vutbr.cz (R. Dobai), korenek@fit.vutbr.cz (J. Korenek), sekanina@fit.vutbr.cz (L. Sekanina).

http://dx.doi.org/10.1016/j.asoc.2017.03.009
1568-4946/

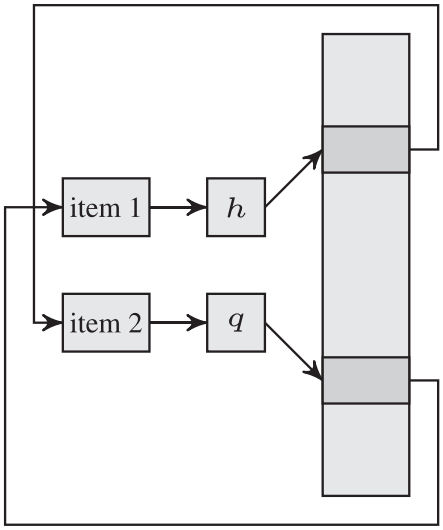

Fig. 1. Cuckoo hashing with two hash functions.

functions and, as a consequence, cuckoo hashing can rearrange the items in the table. It is possible that the same item is pushed-out twice. In this case an unresolvable collision exists and the table must be rehashed with new hash functions just like in the case of perfect hashing. However, the iterative rearrangement gives a higher probability for the insertion to be successful. Perfect hash requires time consuming rehashing more often [5,7].

The work presented in this paper uses cuckoo hashing for FPGA-based IP address filtering. A pipelined reconfigurable hash function with parallel computation is proposed. The proposed evolutionary algorithm (EA) fine-tunes the reconfigurable hash function for the given set of IP addresses selected for filtering/monitoring. The proposed hash function provides the lookup of IP addresses at a speed suitable for high-speed computer networks and achieves a higher table-load factor in comparison with conventional hash functions. As a consequence, the tables are filled up with more IP addresses and the filter will be offline less frequently due to rehashing or table replacement.

The rest of the paper is organized as follows. Sections 2 and 3 introduce the state-of-the-art of hashing in FPGAs and the evolutionary design of hash functions, respectively. Section 4 deals with the proposed FPGA-based system for IP address filtering. The reconfigurable hash function is proposed in Section 5. The experimental results are discussed in Section 6 and the paper is concluded in Section 7.

2. Hashing in FPGA-based network filters

FPGA is a device consisting of universal, reconfigurable elements arranged into a two-dimensional array and reconfigurable interconnections between them. The desired functionality is mapped into various elements: Boolean functions usually into several lookup tables (LUTs), larger memory blocks into collections of block random-access memories (BRAMs) and complex arithmetic operations into digital signal processing (DSP) slices. These reconfigurable elements are interconnected in order to achieve more complex functionality. The configuration for elements and interconnections are assembled into the bitstream and uploaded into the FPGA which result in the given configuration in the hardware.

2.1. Hashing and collisions

There exists no accurate definition of hashes [12] but, in general, a hash function processes a variable-length input and produces the fixed-length output called hash. In hashing tables the hash is used as the address for the item. For example, the hash computed for the given IP address determines its location as the offset from the beginning of the hash table.

More than one input can have the same hash, therefore, two or more IP addresses can be assigned to the same table location. This is called a collision and can have a negative effect on the lookup time because all of these collided IP addresses need to be checked and compared. Collisions occur more frequently when the table utilization is more than 50% [12]. Neither such a low utilization nor increased lookup time are acceptable in high-speed network monitors implemented in FPGAs with limited memory resources.

2.2. Cuckoo hashing and challenges of FPGA implementations

Cuckoo hashing uses more than one hash functions which allows us to achieve higher table utilization without collisions [7]. A collision is resolved by rearranging the table items. Employing more hash functions results in a higher number of records in the table without collision. An unresolvable collision occurs usually with two hash functions when around 50% of the table is already used, around 90% with three and 95% with four hash functions [8,13]. The use of four hash functions, however, requires four memory accesses which might be feasible by only using multiple memory interfaces in parallel because one external memory access requires time very close to what is available for filtering in 400 Gbps networks. Hash tables for IP filters need to be in external memories because of their size. Multiple memory interfaces are available in the latest and most advanced 400 Gbps network cards based on FPGAs [1,2], but just one filter application cannot occupy the whole memory bandwidth because other filtering and monitoring applications are required as well.

It is possible to place some small and limited number of IP addresses into the same table position [14]. However, even two sequential memory accesses for two items in a position can take longer than the time available for the lookup. Furthermore, more items in a table position multiplies the memory requirements and, therefore, very limited internal memory resources like BRAMs cannot be used.

Placing the conflicted IP addresses into another table is also possible [15]. This table can be implemented as content-addressable memory but requires a considerable amount of resources in the FPGA. The capacity of the additional memory is significant, requiring about 1/64 to 1/16 the total capacity of the main hash table [3].

Hash tables with cuckoo hashing use generic hash functions which were developed to work well in general for various types of inputs, but were not optimized for working together for cuckoo hashing. Popular hash functions are the Jenkins hash functions like lookup3, SpookyHash and functions used for cyclic redundancy check (CRC) computation [3,11].

The work presented in this paper considers cuckoo hashing with two functions only in order to limit the memory accesses. Furthermore, the proposed approach requires no memory overhead. The two hashes are computed with two instances of the proposed reconfigurable hash function which are optimized to work well *together* for cuckoo hashing. The goal of the optimization is to improve the table utilization (table-load factor) without introducing any time or hardware overhead.

3. Evolutionary design of hash functions

Since there is no definition for the exact behavior of the hash function one cannot use a deterministic algorithm for development [12]. Characteristics such as output uniform distribution, table-load

factor, collision rate and avalanche effect can be used to evaluate hash functions, but it is not known how to reverse this process and define the function which will have good characteristics for various inputs. General-purpose hash functions used today were developed by experts with years of experience and are based on their intuitions for finding hash functions.

EAs can be successfully used for domains where the system under development cannot be well defined and is partly error resilient, e.g. image filters, packet classifiers and other examples given in [16]. EAs work with a population of candidate solutions. Each solution is evaluated and its fitness is determined. The fitness reflects how good the solution is for solving the given problem. Candidate solutions with better fitness have a better chance to survive. The population is extended by new solutions created from the fittest solutions. A certain number of generations are produced and the population fitness usually improves over time if the EA is tuned well. The search is concluded when a limit is reached, e.g. the time of evolution or the number of generations. The solution with the best fitness is proclaimed as the result of the search.

Hash functions were developed by using EAs, more precisely by genetic programming (GP) [17,18] and grammatical evolution [19]. The variable size of the candidate hash functions makes these approaches inefficient for FPGAs. Cartesian genetic programming (CGP) [20] uses fixed-size candidate solutions which map well into FPGAs. CGP-based hash function evolution was proposed for a packet classifier [21].

State-of-the-art methods are using the avalanche effect [17] as the fitness function or trying to minimize the collision rate [19]. These approaches are aimed at the development of general-purpose hash functions.

The work presented in this paper evolves hash function *pairs* which work well together for cuckoo hashing and for the given set of IP addresses. Therefore, the evolved hash functions are custom tailored and not general purpose.

The preliminary results were published in [22] where a non-linear feedback shift register (NLFSR) was used and optimized as the hash function. However, these functions process the IP address sequentially (bit after bit) and, therefore, the lookup is multiple times slower than that of the parallel computation considered in this paper. The proposed hash function pair significantly improves the table-load factor, as it will be demonstrated later.

4. IP address filtering

The system for IP address filtering is shown in Fig. 2. It consists of three main parts: the software (SW), the FPGA and the external memory (ext). The evolutionary design of hash functions is implemented in SW. The hash function configurations are uploaded into the FPGA where the IP address filtering is performed at high-speed. The hash table containing the desired set of IP addresses is in the external memory (ext).

The system is designed in such a way that it can be integrated to commercially available monitoring solutions [2].

4.1. IP address lookup in the FPGA

The IP address lookup is implemented in the FPGA in order to achieve processing speed necessary for high-speed computer networks. The steps of the lookup are highlighted in Fig. 2: (a) The content of the configuration register sets the configuration of the hash function pair via multiplexers. The content can be even random in the beginning because the hash table is still empty and low table utilization can be achieved even with a “bad” hash function. (b) IP addresses selected for filtering/monitoring are sent into the FPGA and (c) inserted into the hash table by cuckoo

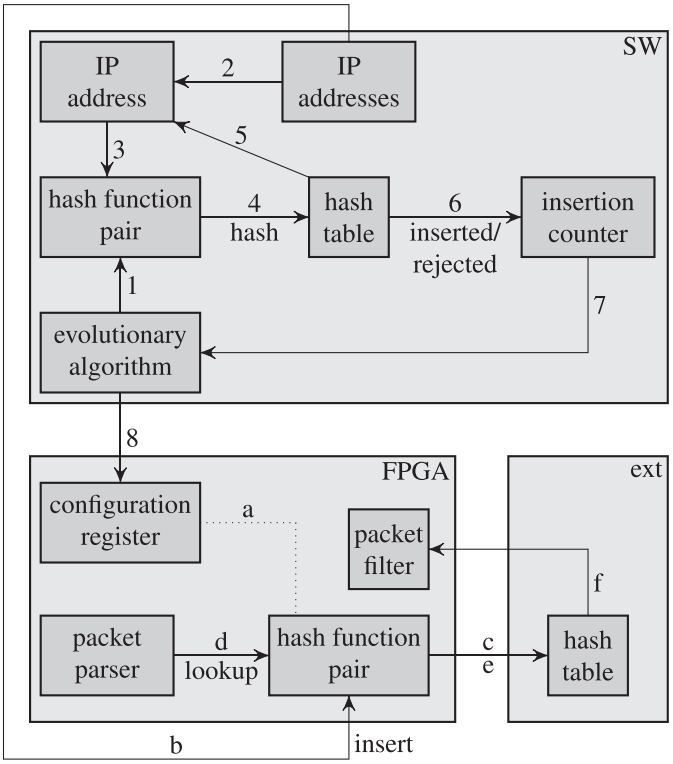

Fig. 2. IP address filtering with the evolution of new hash functions.

hashing. The insertion has a linear complexity because cuckoo hashing may rearrange the previous contents prior to inserting a new IP address. This is acceptable since network filters require fast and uninterrupted lookup but can tolerate longer insertion times. (d) Packet processing starts with the parsing and extraction of the IP address. (e) The source IP address is looked-up in the hash table. (f) The packet filter either drops the packet or performs logging/monitoring if there is a match of the source IP address in the hash table.

It can be desirable to optimize the hash functions for table utilization after a time. The optimization is executed by the SW in regular intervals and the resulted new hash functions are uploaded into the FPGA. The upload is fast and is a matter of only a couple of milliseconds because the IP addresses are pre-processed in the SW. The network traffic can be stored in buffers when the traffic is slow and the hash tables can be updated in the filter without any packet loss.

4.2. Evolution of new hash function configurations

The goal of the evolution is to optimize the hash function pair for the high table-load factor, i.e. to achieve that new IP addresses could be inserted into the table, which was not possible with the current hash functions. The evolution might ensure that the filter could be extended with new IP addresses and delay the moment when the IP filter needs to be replaced with hash tables of a higher capacity.

The evolutionary optimization shown in Fig. 2 takes place as follows. (1) The whole process is guided by the EA which works with a population of candidate hash function pairs. (2) The IP addresses which are already in the filter are supplemented by addresses which cannot be inserted into the hash table because of the limitation of the hash functions in the filter. The new set of IP addresses is used for training new candidate hash functions. (3) Each IP address is hashed and (4) inserted into the hash table in the SW-based

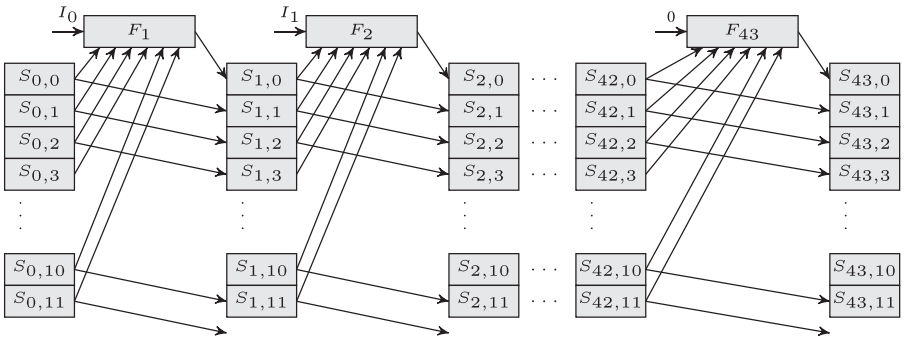

Fig. 3. Proposed pipelined reconfigurable hash function component with parallel computation.

simulator. (5) The IP address pushes-out the previous occupant at the given table position which is rehashed into another position with the another hash function. Steps (3–5) are repeated until all the collisions are resolved or an unresolved collision is detected. An unresolved collision occurs if the first IP address is pushed-out again from the table. (6) If the collisions are resolved then the counter of the successful insertions is incremented and the process of inserting a new IP address starts from step (2). (7) In the case of an unresolvable solution, the content of the counter is used by the EA as the fitness of the candidate hash function pair. The process of evaluating other candidate functions follows from step (1). (8) The evolution is concluded after a pre-selected number of candidate solutions is generated and evaluated. The configuration of the hash function pair which achieved the best fitness is uploaded into the FPGA. New IP addresses can now be added into the hash table in the FPGA starting with step (b) after which the new IP addresses can be found in the table and the packets are filtered/monitored.

4.3. Evolutionary algorithm

Candidate hash functions are encoded using bitstrings which define the configurations for configurable hash function components.

The evolution starts with a random population of λ candidate hash function pairs. The candidate hash functions are evaluated and their fitness is determined. The hash function pair with the best fitness is selected and the other functions are discarded. The population of λ individuals is completed by creating $\lambda - 1$ offspring solutions from the fittest hash function pair. An offspring is spawned by copying the chromosome of the parent and making several random modifications, *mutations* in the chromosome.

This process of selection and reproduction is repeated until a desired number of candidate hash function pairs is generated and evaluated.

4.4. Fitness

Candidate hash function pairs are evaluated as follows. IP addresses are inserted into the hash table until an unresolvable collision is detected, i.e. the IP address cannot be inserted into the table. The number of successful insertions is used as the fitness of the candidate hash function pair.

This measurement reflects how well the function pair works together and it directs the search towards pairs which achieve together a high table-load factor.

5. Reconfigurable hash functions for FPGA-based IP filters

In general, a hash function is a Boolean function $h : B^x \rightarrow B^y$ where x is the number of input bits and y the number of bits in the output hash.

The hash function for an IP address version 4 requires the processing of $x = 32$ inputs. The size of the output depends on the required capacity of the hash table but it can be assumed that it has at least $y = 10$ bits. Therefore, the problem of the hash function pair design can be classified as hard.

A pipelined reconfigurable hash function component with parallel computation is proposed in this paper. The hash function was developed with the goal of being parallel, i.e. process all the inputs at once; and to be pipelined (i.e. produce a hash in each clock cycle). Another goal was to help the search by only allowing such possible configurations which are all good for hashing with a reasonable probability.

The proposed reconfigurable hash function component is shown in Fig. 3 where $I_0, \ldots, I_{31}$ are the bits of the IP address, $F_1, \ldots, F_{43}$ are reconfigurable function-blocks, $S_{i,j}$ are 1-bit registers for all $j \in \{0, \ldots, 12-1\}$ and all $i \in \{0, \ldots, 43\}$; all under the assumption that 12-bit hashes are computed. Actually, two hash functions are used for a hash function pair and the hash table has a capacity of $2 \times 2^{12} = 8$k. The seed (initial value) of the hash is $(S_{0,11}, \ldots, S_{0,0})$ and the result of the hash computation is $(S_{43,11}, \ldots, S_{43,0})$.

The seed is propagated through 43 stages of the hash function and is mixed with each input bit of the IP address. 32 stages are used for processing the 32-bits of the IP address and additional $12-1$ zeros are mixed into the hash value in $12-1$ stages, where 12 is the length of the hash in the example. These "zero stages" are used to enforce one of the characteristic properties of good hash functions: each input should have an influence on all of the outputs. Since the last processed bit of the IP I_{31} influenced only $S_{32,0}$, therefore, $12-1$ additional stages are used in order to propagate the bit to $S_{32+12-1,0+12-1} = S_{43,11}$.

5.1. Processing in a stage

Each processing stage $i+1$ performs two operations based on bits $S_{i,j}$ of the previous state and input bit I_i:

1. The state bits $S_{i,j}$ are shifted in the direction from $j = 0$ to $j = 11$ by one position and $S_{i,11}$ is discarded.
2. Function block F_{i+1} computes $S_{i+1,0}$ as the combination of the previous state bits $S_{i,j}$ and input bit I_i.

The result is the new state $(S_{i+1,0}, \ldots, S_{i+1,11}) = (S_{i+1,0}, S_{i,1}, S_{i,2}, \ldots, S_{i,10}) = (F_{i+1}, S_{i,1}, S_{i,2}, \ldots, S_{i,10})$ by using the shifted previous state $(S_{i,1}, \ldots, S_{i,10})$ and F_{i+1} as described previously in steps 1 and 2, respectively.

Function blocks F_{i+1} are reconfigurable as shown in Fig. 4 and used as follows. Configuration bits $M_{i,0}, \ldots, M_{i,11}$ enable/disable by AND gates the state bits $S_{i,0}, \ldots, S_{i,11}$ in the XOR gate. For example, if $M_{i,0} = 0$ then $S'_{i,0} = 0$ and $S_{i,0}$ does not influence the results, or else

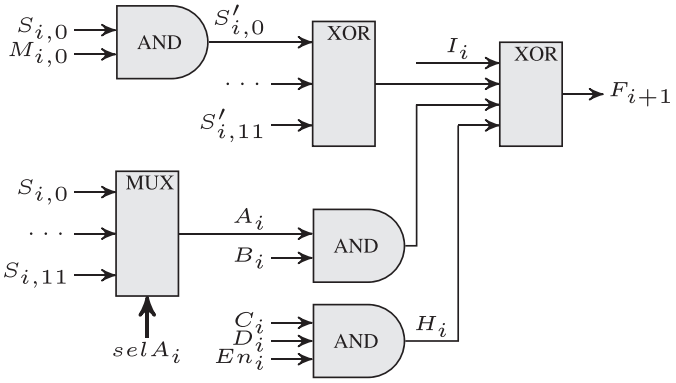

Fig. 4. Reconfigurable function-block for stage $i+1$ in the pipelined hash function.

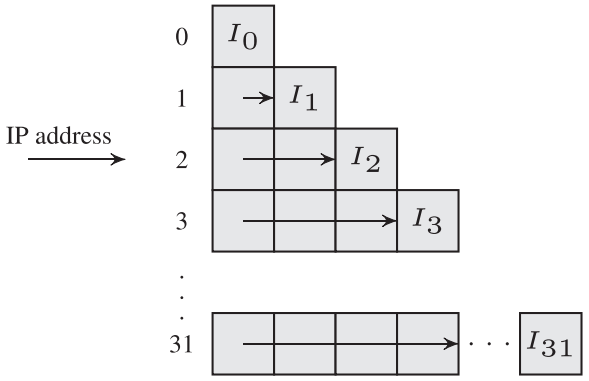

Fig. 5. Input sequencer for the pipelined hash function.

if $M_{i,0}=1$ then $S'_{i,0}=S_{i,0}$ and therefore $S_{i,0}$ will be applied to the output through the XOR gates.

The function block applies one or two AND operations to the state inputs. These are also included in the final result F_{i+1} through the XOR gate. The states are selected for these operations by the use of multiplexers (MUXs) and configuration inputs $selA_i, \ldots, selD_i$. For example, if $selA_i=0$ then $A_i=S_{i,0}$, if $selA_i=11$ then $A_i=S_{i,11}$. The second, optional AND gate can be turned on/off by En_i. For example, if $En_i=0$ then $H_i=0$, or if $En_i=1$ then $H_i=C_i \wedge D_i$.

5.2. Chromosome for representing the candidate solution

A candidate solution is the configuration of the hash function pair and is represented by the chromosome in the EA. The chromosome contains the following items for both hash functions: $M_{i,0}, \ldots, M_{i,12-1}$, $selA_i$, $selB_i$, $selC_i$, $selD_i$, En_i shown in Fig. 4 where $M_{i,0}, \ldots, M_{i,12-1}, En_i \in \{0, 1\}$ and $selA_i, selB_i, selC_i, selD_i \in \{0, \ldots, 12-1\}$ for all $i \in \{0, \ldots, 32-1+12-1\}$ considering 32-bit inputs and 12-bit hash functions.

5.3. Search space

The search space is significantly reduced because it does not allow us to develop arbitrary Boolean functions. All of the configurations represent good mixing functions of the inputs with the initial seed. The EA can be more successful in this limited search space than if it would have to develop hash functions from scratch.

State bits $S_{i,j}$ are shifted in the direction from $j=0$ to $j=11$. On the other hand, F_{i+1} mixes some of the state bits and puts them to the beginning of the shift. Therefore, input bits propagate in both directions and are well mixed with the state bits. A more general approach would be to develop functions individually for all state bits, i.e. develop 43×12 function blocks instead of the current 43. It is obvious that the search space would be much larger and the search will be probably less successful. The use of shifting and only one function-block per stage ensure a simpler complexity of the search, but also good mixing at the same time.

This behavior is motivated by NLFSRs. A Fibonacci-type NLFSR is a shift register with a feedback function containing XOR and AND operations over the state bits. However, NLFSR is sequential, i.e. produces one input in each clock cycle. The parallel version of NLFSR with the restriction of $F_1=F_2=\cdots=F_{43}$ would be equivalent to the proposed hash function. By allowing for the inequality between the function-blocks of the stages, one can develop a hash function which can be described as parallel NLFSR with changing feedback functions in each clock cycle.

The proposed hash function component includes another structural constraint in order to limit the search space. Feedback functions of NLFSRs with a maximal period mostly contain only one or two AND operations between states [23]. This observation motivated the structure of the proposed function-block which contains one mandatory AND and a second which is optional and is enabled/disabled by En_i as can be seen in Fig. 4.

5.4. Pipelined hash computation

Fig. 5 shows the input sequencer for the pipelined hash function. A 32-bit IP address can be inserted into the sequencer and the previous addresses are shifted by one position in each clock cycle. The outputs $I_0, \ldots, I_{31}$ in Fig. 5 are interconnected with the inputs in Fig. 3 while I_0 is delayed for 0 clock cycle, I_1 by 1 cycle, ..., and I_{31} by 31 cycles. The reason for these delays of various sizes is that the inputs are required in various sequential depths in the proposed hash function component.

The input sequencer, together with the multi-stage implementation of the hash function, ensures that an IP address can be sent-in in each clock cycle. The first hash will be computed with an initial latency of 43 cycles, but after that a hash will be produced in each clock cycle.

5.5. Comparison with conventional hash functions

Conventional general-purpose hash functions usually produce 32-, 64-, or 128-bit hashes. Consequently, the bit precision is reduced to the size required by the hash table. One can select some of the bits, or can combine them by XOR folding [24]. This gives lots of possibilities and it is not known which will produce the best results. The proposed hash function component produces hashes with bit precision matching the requirements of the hash table and, therefore, one does not need to address the transformation of larger hashes to the desired precision. The design process controlled by the EA guides the development of the hash function pair and handles the transformation of the hashed key into the hash of the desired precision.

It cannot be guaranteed that the results will be good, even with the best general-purpose functions. These functions were developed to work well alone. The work presented in this paper considers hash function pairs which are optimized to work well together for cuckoo hashing.

The proposed hash function component is reconfigurable and can be re-evolved when necessary. Alternatively, one can implement various conventional hash functions and evaluate them during rehashing. However, the proposed reconfigurable hash function component offers many more possible configurations in comparison with the approach based on several manually implemented conventional hash functions.

6. Experimental results

The proposed reconfigurable hash function component, together with the EA-based automated tuning for selected IP addresses, were implemented and evaluated. The FPGA-based fast lookup of IP addresses was investigated in a XC7Z020 Zynq all programmable (AP) system-on-chip (SoC) device and the SW-based evolution in an Intel Xeon E5-2630 processor. The IP addresses used in the experiments were extracted from a firewall in the Czech national research and education network (CESNET). Hash tables with a capacity of 8k records or more were considered. The experiments were statistically evaluated based on 30 independent runs. The implementations of conventional hash functions were obtained from [25].

The EA works with $\lambda = 5$ candidate hash function pairs in each generation. The size of the population was selected as the value common for various applications [16,20,26,27] and the fact that it seems to have a negligible influence on the quality of the final solutions considering a constant number of generated and evaluated candidate solutions.

6.1. Tuning parameters for EA

The experiments revealed that the quality of evolved solutions can be improved by employing two different kinds of mutations during evolution. Coarse-grained mutation enforces and conserves the equality between the configurable function blocks, i.e. $F_1 = F_2 = \cdots = F_{43}$. This is ensured by mutating only F_1 and copying the part of the chromosome of F_1 into F_2, F_3, ..., F_{43}. Fine-grained mutation changes the configurable function blocks individually, i.e. the function blocks can be different: $F_1 \neq F_2 \neq \cdots \neq F_{43}$. Coarse-grained mutation supports fast convergence in the beginning and this is followed by fine-grained mutation for further improvement of the candidate solutions. It should be noted that mutation is not binary, it works at the level of chromosome components.

Coarse-grained mutation is performed in the first 20k generations of candidate hash function pairs and is followed by the fine-grained mutation considering 200k total number of generations. These values were determined after observing the fitness development and switching to fine-grained mutation or concluding the search after the fitness did not improve significantly in consequent generations.

Fig. 6 shows the impact of the number of coarse-grained mutations on the fitness. The box plot of 30 independent runs shows the fitness value of the best candidate solution, i.e. the number of successfully inserted IP addresses in a 8k hash table. The fitness improves by increasing the number of mutations and the variation between runs become less significant. The number of mutations per chromosome of around 8 seems to be the optimal value.

Fig. 7 shows the impact of fine-grained mutations on the fitness. It can be concluded that a too high mutation rate can be counterproductive and the number of mutations per chromosome of around 3 seems to improve most the candidate solutions.

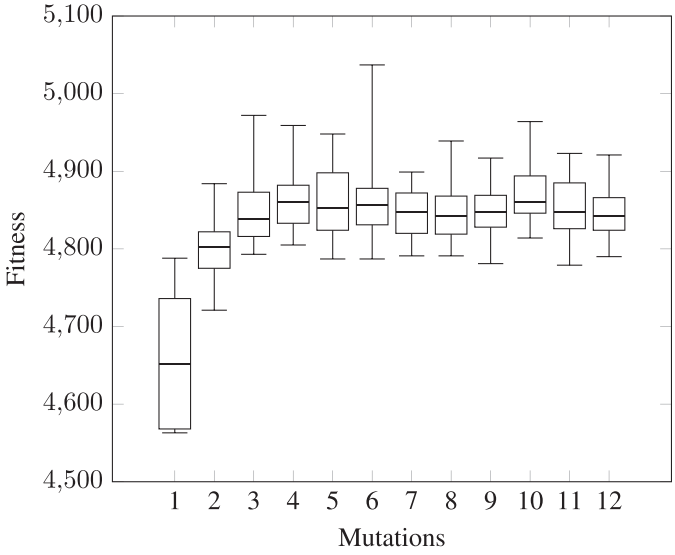

Fig. 6. Impact of coarse-grained mutations on the fitness.

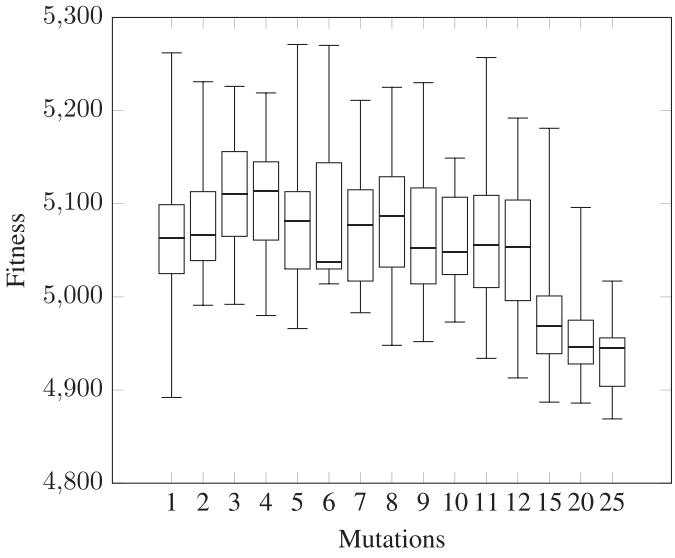

Fig. 7. Impact of fine-grained mutations on the fitness.

6.2. Comparison with conventional and unconventional solutions

The proposed pipelined reconfigurable hash function component was compared with conventional hash functions and the state-of-the-art unconventional CGP as well. The unconventional solution was created as the GP-based approach [17], reimplemented as CGP with constant-size candidate solutions and the same fitness function that was considered for the proposed approach in this paper. The CGP approach develops hash functions very similar to the conventional ones because it uses the same elementary operations. The developed hash functions are sequential, therefore, the lookup is much slower than in the case of the proposed hash function component.

Conventional hash functions with the support for seed values and 32-bit hashes were selected. The hash function pair was created by using the same function with seeds 0 and 1, just like in the case of the proposed hash function pair.

Table 1 shows the achieved results where the solutions marked with (c) are created by cropping the 32-bit hashes and using the least significant bits; and for the solutions with (f) the number of hash bits were reduced using XOR folding. The table contains the results for the best, the upper quartile (75%), the median (med.) and the lower quartile (25%) of the unconventional solutions. The number of successfully inserted IP addresses (Ins) and the table utilization (%) for the 8k hash table are shown in the table for four different IP address sets (Set 1–4) extracted from a firewall in the backbone network of CESNET.

The cropped version of lookup3 achieves the best result for Set 1 amongst the conventional hash functions. The folded MurmurHash3 is the best for Set 2, the cropped MurmurHash3 for Set

Table 1
Comparison of inserted records with conventional hash functions.

Function	Set 1		Set 2		Set 3		Set 4	
	Ins	%	Ins	%	Ins	%	Ins	%
proposed (best)	5226	63.79	5235	63.9	5261	64.22	5173	63.15
proposed (75%)	5156	62.94	5132	62.65	5139	62.73	5121	62.51
proposed (med.)	5111	62.39	5081	62.02	5099	62.24	5070	61.88
proposed (25%)	5065	61.83	5036	61.47	5046	61.6	5044	61.57
CGP (best)	4922	60.08	4868	59.42	4872	59.47	4895	59.75
CGP (75%)	4799	58.58	4817	58.8	4829	58.95	4848	59.18
CGP (med.)	4783	58.39	4788	58.45	4804	58.64	4817	58.8
CGP (25%)	4758	58.08	4771	58.24	4768	58.2	4790	58.47
CRC32 (c)	3674	44.85	3425	41.81	3401	41.52	4176	50.98
MurmurHash3 (c)	4199	51.26	3827	46.72	4179	51.01	3384	41.31
MurmurHash3 (f)	3365	41.08	4364	53.27	3839	46.86	3074	37.52
SpookyHashV2 (c)	3528	43.07	3449	42.1	4121	50.31	2176	26.56
SpookyHashV2 (f)	3759	45.89	4260	52	4062	49.58	2128	25.98
lookup3 (c)	4516	55.13	4047	49.4	3996	48.78	3951	48.23
lookup3 (f)	4140	50.54	3656	44.63	3702	45.19	2914	35.57
fnv-1a (c)	3787	46.23	2926	35.72	1995	24.35	2583	31.53
fnv-1a (f)	3223	39.34	3557	43.42	2326	28.39	3949	48.21

3 and CRC32 for Set 4. It is obvious that one needs to implement all of these hash functions in order to be able to select the best one for the given IP address set.

Conventional solutions achieve a table utilization of 30–55%. The unconventional CGP reaches up to 58–60% and the proposed solutions over 63%. The EA is well tuned and, therefore, even the 25% quartile gives results of around 62%. One can select the best evolved solution of the runs because the evolution is executed in parallel with the FPGA-based lookup.

The fitness development for Set 1 is shown in Fig. 8. Unconventional solutions need just a couple of hundred generations for outperforming lookup3, the best conventional solution. CGP performs well, but the proposed hash function even with random search achieves higher fitness. It should be noted that it is not a pure random search since it uses the proposed representation which reduces the search space significantly. Further improvement in the quality of solutions can be observed in the case of the proposed EA (fine-grained). The evolution with the coarse-grained mutation seems to start stagnating around 0.2×10^5 generations. The mutation is switched to fine-grained at that point which results in further rapid improvement of the fitness. It can be concluded that the evolution with coarse- and fine grained mutations achieves better results than by just using fine-grained mutation from the beginning.

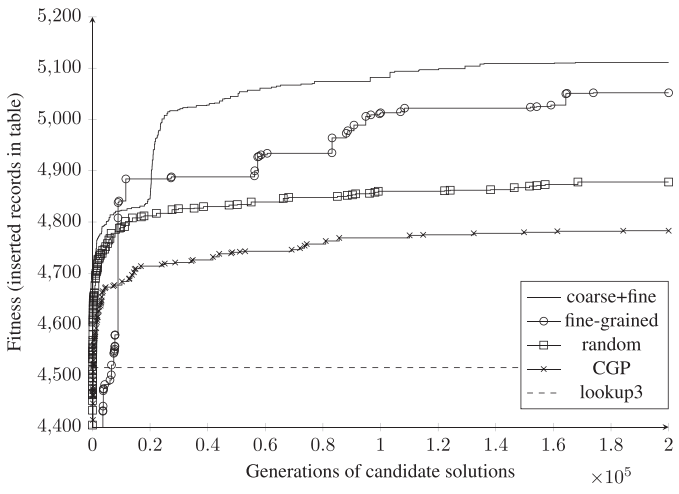

Fig. 8. Comparison of fitness development (median values).

The proposed EA fine-tunes the reconfigurable hash function and this allows for the storing of a couple of hundred more IP addresses in the hash table of the same size. This can be very useful in practice because the IP filter can be used longer without redesigning the whole system with larger hash tables.

6.3. Time required for evolution

The proposed hash function component computes 12-bit hashes in 43 clock cycles for 32-bit inputs. The implemented FPGA-based IP filter can be operated at 260 MHz which is limited by the critical timing path through the reconfigurable function-blocks. The first hash is computed with an initial latency of $1000/260 \times 43 \approx 165$ ns. Consequent hashes are produced in each clock cycle because an IP address can be sent into the hash function in each clock cycle. This gives 260 million hashes per second, and a hash in less than $1000/260 \approx 4$ ns which is fast enough for 400 Gbps networks.

Table 2 contains the achieved table utilization and average evolution time for various table sizes. An 8k set was prepared for the 8k table, a 16k set for the 16k table where the IP addresses from the 8k set were reused and extended with another 8k IP addresses, etc. The 128k set for the 128k table contains the data from the 64k set and another 64k new IP addresses.

It can be observed that the quality of solutions becomes lower with larger tables. This is caused by the fact that a collision is more probable with more IP addresses and the probability of resolving these collisions is lower. A table utilization of 54.4% for a 128k table is still good when one considers the results for conventional hash functions (in the order as they are listed in Table 1): 41.03%, 31.94%, 16.15%, 49.84%, 48.74%, 38.06%, 48.80%, 29.77%, 50.42%.

A larger IP address set requires more time for evaluating the candidate solutions but it does not influence the search space because the IP addresses remain the same size. Therefore, the time required for evolution increases linearly with the size of the hash table. Migration to IP addresses of version 6 will have a negative impact, but currently only addresses of version 4 are used in the network filter.

The 8k hash table provides enough filtering rules, but even hash functions for larger tables can be evolved in a reasonable time. A couple of hours are available for example in the night when the network traffic is slow. It should be noted that parallel candidate solution evaluation is possible and the time for the evolution can be reduced significantly if several processors (computers) are used together.

Table 2
Table utilization and average evolution time for larger hash tables.

8k set		16k set		32k set		64k set		128k set	
%	h	%	h	%	h	%	h	%	h
63.79	1.51	60.8	3.99	58.21	6.12	55.98	7.89	54.4	16.7

Hash 1:

$$F_1 = I_0 \oplus (S_{0,9} \wedge S_{0,11}) \oplus S_{0,0} \oplus S_{0,1} \oplus S_{0,2} \oplus S_{0,6} \oplus S_{0,8} \oplus S_{0,10} \oplus S_{0,11}$$

$$F_2 = I_1 \oplus (S_{1,5} \wedge S_{1,8}) \oplus (S_{1,2} \wedge S_{1,7}) \oplus S_{1,0} \oplus S_{1,2} \oplus S_{1,3} \oplus S_{1,4} \oplus S_{1,5} \oplus S_{1,6} \oplus S_{1,7} \oplus S_{1,8} \oplus S_{1,9} \oplus S_{1,10} \oplus S_{1,11}$$

$$F_3 = I_2 \oplus (S_{2,8} \wedge S_{2,11}) \oplus S_{2,0} \oplus S_{2,4} \oplus S_{2,5} \oplus S_{2,7} \oplus S_{2,8} \oplus S_{2,10}$$

$$\vdots$$

$$F_{43} = 0 \oplus (S_{42,4} \wedge S_{42,7}) \oplus S_{42,0} \oplus S_{42,1} \oplus S_{42,4} \oplus S_{42,6} \oplus S_{42,7} \oplus S_{42,9} \oplus S_{42,10} \oplus S_{42,11}$$

Hash 2:

$$G_1 = I_0 \oplus (T_{0,3} \wedge T_{0,8}) \oplus T_{0,0} \oplus T_{0,1} \oplus T_{0,2} \oplus T_{0,7} \oplus T_{0,10} \oplus T_{0,11}$$

$$G_2 = I_1 \oplus (T_{1,2} \wedge T_{1,6}) \oplus T_{1,0} \oplus T_{1,1} \oplus T_{1,3} \oplus T_{1,4} \oplus T_{1,5} \oplus T_{1,7} \oplus T_{1,9} \oplus T_{1,10}$$

$$G_3 = I_2 \oplus (T_{2,4} \wedge T_{2,10}) \oplus (T_{2,6} \wedge T_{2,10}) \oplus T_{2,0} \oplus T_{2,1} \oplus T_{2,3} \oplus T_{2,7} \oplus T_{2,8} \oplus T_{2,9} \oplus T_{2,10}$$

$$\vdots$$

$$G_{43} = 0 \oplus (T_{42,6} \wedge T_{42,8}) \oplus (T_{42,1} \wedge T_{42,7}) \oplus T_{42,0} \oplus T_{42,3} \oplus T_{42,4} \oplus T_{42,6} \oplus T_{42,7} \oplus T_{42,8} \oplus T_{42,11}$$

Fig. 9. Example of an evolved candidate solution.

6.4. Example evolved hash function pair

An example of an evolved hash function pair is partially shown in Fig. 9 where *F* and *G* are functions for the function-blocks, and *S* and *T* are state bits for the two hash functions. The functions are seeded with values 0 and 1. The evolved hash function pair achieves 63.79% table utilization.

6.5. Required FPGA resources

The FPGA-based IP address filter was implemented using the Vivado tool with *FlowPerfOptimizedHigh* synthesis and *PerformanceExplore* implementation options. The required resources for the reconfigurable hash function are compared with the 32-bit parallel version of lookup3 in Table 3.

Lookup3 was implemented manually based on the software code. It requires 32-bit operations and registers. The hash computation is done using seven operations of XOR, rotations and subtractions, and one addition. The most area consuming operations are the subtractions and additions.

The proposed hash function component uses only simple logic gates. The area requirements are significant mainly because of implementing the reconfigurability in the functions-blocks. As a result, the implementation requires three times more LUTs and two times more registers than lookup3. The increased number of registers is caused by the processing of more stages (more cycles). The initial latency is three times longer, but the following IP addresses can be computed at the same speed (one in each clock cycle).

Table 3
Comparison of implementations in Zynq AP SoC

	lookup3	Evolved	
			Ratio
Clock cycles	16	44	2.75
Slice LUTs	358	951	2.66
Slice registers	1033	1664	1.61
Multiplexers	0	94	

The proposed hash function component requires more FPGA resources for implementation in comparison with conventional hash functions. However, the proposed hash function can be reconfigured and fine-tuned for the given set of IP addresses. On the other hand, one needs to implement several conventional hash functions and switch between them manually.

7. Conclusions

Optimization of cuckoo hashing for FPGA-based IP address filtering was presented in this paper. The proposed pipelined reconfigurable hash function component with parallel computation is fine-tuned by the EA for the given set of IP addresses selected for filtering/monitoring. The proposed hash function component provides the lookup of IP addresses at a speed suitable for high-speed computer networks because with an initial latency it is able to produce hashes in each clock cycle.

It is not necessary to stop the IP filter in order to optimize the hash functions. The optimization can be performed when the network traffic is slow and without any loss of packets.

The EA optimizes the hash configuration for high table utilization. The experiments showed that the utilization can be improved by 10% or more in comparison with even the best conventional hash function.

Multiple external memory accesses decrease the throughput of the IP filter because lookup takes longer. Single access results in a high-speed, but at the cost of low table utilization. The proposed EA-based approach is able to increase the utilization without sacrificing the speed of the lookup.

Interesting future research includes the migration to IP addresses of version 6 and is considering other uses of the proposed hash function component in the field of network filtering.

Acknowledgment

This work was supported by The Ministry of Education, Youth and Sports of the Czech Republic from the National Programme of

Sustainability (NPU II); project IT4Innovations excellence in science – LQ1602.

References

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

[2] L. Kekely, J. Kucera, V. Pus, J. Korenek, A.V. Vasilakos, Software defined monitoring of application protocols, IEEE Trans. Comput. 65 (2) (2016) 615–626, http://dx.doi.org/10.1109/TC.2015.2423668.

[3] L. Kekely, M. Zadnik, J. Matousek, 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, http://dx.doi.org/10.1109/DDECS.2014.6868793.

[4] D. Tong, Y.-H.E. Yang, V.K. Prasanna, A memory efficient IPv6 lookup engine on FPGA, in: International Conference on Reconfigurable Computing and FPGAs, ReConFig, 2012, 2012, http://dx.doi.org/10.1109/ReConFig.2012.6416760, art. no. 6416760.

[5] V. Pus, J. Korenek, Fast and scalable packet classification using perfect hash, in: 7th ACM SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA'09, 2009, pp. 229–235, http://dx.doi.org/10.1145/1508128.1508163.

[6] D. Belazzougui, F.C. Botelho, M. Dietzfelbinger, Hash, displace, and compress, in: 17th Annual European Symposium on Algorithms, ESA 2009, Vol. 5757 of Lecture Notes in Computer Science, 2009, pp. 682–693, http://dx.doi.org/10.1007/978-3-642-04128-0_61.

[7] R. Pagh, F.F. Rodler, Cuckoo hashing, in: Algorithms – ESA 2001, Vol. 2161 of Lecture Notes in Computer Science, 2001, pp. 121–133, http://dx.doi.org/10.1007/3-540-44676-1_10.

[8] S. Pontarelli, P. Reviriego, J.A. Maestro, Parallel d-pipeline: a cuckoo hashing implementation for increased throughput, IEEE Trans. Comput. 65 (1) (2016) 326–331, http://dx.doi.org/10.1109/TC.2015.2417524.

[9] T.N. Thinh, S. Kittitornkun, Massively parallel cuckoo pattern matching applied for NIDS/NIPS, in: Fifth IEEE International Symposium on Electronic Design, Test and Application, 2010, pp. 217–221, http://dx.doi.org/10.1109/DELTA.2010.46.

[10] L. Kekely, V. Pus, J. Korenek, Software defined monitoring of application protocols, in: 2014 Proceedings IEEE INFOCOM, 2014, pp. 1725–1733, http://dx.doi.org/10.1109/INFOCOM.2014.6848110.

[11] M. Dvorak, J. Korenek, Low latency book handling in FPGA for high frequency trading, in: 17th International Symposium on Design and Diagnostics of Electronic Circuits & Systems, 2014, pp. 175–178, http://dx.doi.org/10.1109/DDECS.2014.6868785.

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

[13] D. Fotakis, R. Pagh, P. Sanders, P.G. Spirakis, Space efficient hash tables with worst case constant access time, in: Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science, Vol. 2607 of Lecture Notes in Computer Science, 2003, pp. 271–282.

[14] R. Panigrahy, Efficient hashing with lookups in two memory accesses, in: Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms, 2005, pp. 830–839.

[15] A. Kirsch, M. Mitzenmacher, U. Wieder, More robust hashing: cuckoo hashing with a stash, in: 16th Annual European Symposium on Algorithms, ESA, Vol. 5193 of Lecture Notes in Computer Science, 2008, pp. 611–622, http://dx.doi.org/10.1007/978-3-540-87744-8_51.

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

[17] C. Estebanez, Y. Saez, G. Recio, P. Isasi, Automatic design of noncryptographic hash functions using genetic programming, Comput. Intell. 30 (4) (2014) 798–831, http://dx.doi.org/10.1002/coin.12033.

[18] 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, http://dx.doi.org/10.1145/1570256.1570396.

[19] P. Berarducci, D. Jordan, D. Martin, 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.

[20] J.F. Miller, Cartesian Genetic Programming, Springer Berlin Heidelberg, 2011, http://dx.doi.org/10.1007/978-3-642-17310-3.

[21] H. Widiger, R. Salomon, D. Timmermann, Packet classification with evolvable hardware hash functions – an intrinsic approach, in: 2nd International Workshop on Biologically Inspired Approaches to Advanced Information Technology, Vol. 3853 of Lecture Notes in Computer Science, 2006, pp. 64–79, http://dx.doi.org/10.1007/11613022_8.

[22] R. Dobai, 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, http://dx.doi.org/10.1109/SSCI.2015.174.

[23] E. Dubrova, A list of maximum period NLFSRs, in: Cryptology ePrint Archive: Report 2012/166, 2012 http://eprint.iacr.org/2012/166 (accessed 15.08.16).

[24] FNV Hash. http://www.isthe.com/chongo/tech/comp/fnv/ (accessed 15.08.16).

[25] SMHasher, A Test Suite Designed to Test the Distribution, Collision, and Performance Properties of Non-Cryptographic Hash Functions. https://github.com/aappleby/smhasher (accessed 15.08.16).

[26] R. Dobai, L. Sekanina, Low-level flexible architecture with hybrid reconfiguration for evolvable hardware, ACM Trans. Reconfig. Technol. Syst. 8 (3) (2015), http://dx.doi.org/10.1145/2700414, art. no. 20.

[27] Z. Vasicek, L. Sekanina, Hardware accelerator of cartesian genetic programming with multiple fitness units, Comput. Inform. 29 (6) (2010) 1359–1371.