Evolutionary Design of Fast High-quality Hash Functions for Network Applications

David Grochol
Brno University of Technology
Faculty of Information Technology
IT4Innovations Centre of Excellence
Bozetechova 2
Brno, Czech Republic
igrochol@fit.vutbr.cz

Lukas Sekanina
Brno University of Technology
Faculty of Information Technology
IT4Innovations Centre of Excellence
Bozetechova 2
Brno, Czech Republic
sekanina@fit.vutbr.cz

ABSTRACT

High speed networks operating at 100 Gbps pose many challenges for hardware and software involved in the packet processing. As the time to process one packet is very short the corresponding operations have to be optimized in terms of the execution time. One of them is non-cryptographic hashing implemented in order to accelerate traffic flow identification. In this paper, a method based on linear genetic programming is presented, which is capable of evolving high-quality hash functions primarily optimized for speed. Evolved hash functions are compared with conventional hash functions in terms of accuracy and execution time using real network data.

CCS Concepts

•**Networks** → *Network monitoring;* •**Computing methodologies** → *Search methodologies; Genetic programming;*

Keywords

Linear Genetic Programming, Network applications, Hash function

1. INTRODUCTION

We are witnessing a significant progress in the development of high speed computer networks. Data centers are running at 10 gigabit-per-second (Gbps) speeds and moving to 40 Gbps. Solutions for 100 Gbps are already available. New network applications, new security threats and the growing communication speeds are major current challenges for precise and accurate *network monitoring*. As it turns out that networks have to be monitored at the application layer, it is crucial to identify the application (or the application protocol) which the traffic belongs to [24]. The current practice in the area of network monitoring is based on *flow* measurements, where the flow is uniquely identified by five parameters within a certain time period: source and destination IP address, source and destination port and transport protocol. This means that each packet has to be processed. In order to identify the application (or the application protocol) the network traffic belongs to, one has to inspect one or several packets with a payload. The main difficulty is that the time to process one packet is less than 7 ns in the case of modern 100 Gbps links.

The most promising approach capable of solving this problem is *software defined monitoring* (SDM) [16]. The idea of SDM is that most traffic can be processed in hardware using relatively simple (ad so fast) logic circuits whose functionality (i.e. the rules of operation) can be controlled from software. Unrecognized traffic, which in practice represents only a fraction of the whole traffic, is then analyzed by sophisticated algorithms in software. According to [16], about 80% of flows can be processed in hardware after a learning phase of the SDM system is finished. However, during the learning phase, the software has to handle most of the flows.

One of the most frequently called functions from the software implementation is a *hash function*, which assigns a memory address (slot) where the data of a given flow are stored to the input flow. A good hash function should exhibit some properties (see more in Section 2.1), in particular, the number of collisions have to be minimal for the data of a given target domain. In the case of SDM, there is another important requirement—obtaining of the hash (i.e. the output of the hash function) has to be very fast. The reason is that even if most of traffic is processed in hardware, a relatively intensive data stream (about 20 Gbps) has still to be processed in software. Moreover, the hash function is typically called several times in order to obtain desired address because the memory addressing system is designed as hierarchical, for example, in the cuckoo hashing scheme [22]. Hence it is important to optimize not only the quality of hashing, but also the execution time.

The goal of the paper is to propose and evaluate a method capable of providing high quality and easy-to-compute hash functions for SDM. As hash functions are sequences of instructions, it is natural to utilize linear genetic programming (LGP) for their design. In order to minimize the execution time, candidate hash functions are constructed using simple instructions such as addition and logic operations. LGP is implemented as a parallel evolutionary algorithm exploiting the island model, i.e. there are several independent popu-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

GECCO '16, July 20-24, 2016, Denver, CO, USA
© 2016 ACM. ISBN 978-1-4503-4206-3/16/07…$15.00
DOI: http://dx.doi.org/10.1145/2908812.2908825

lations evolved separately that are exchanging some genetic material according to a predefined pattern. Evolved hash functions are analyzed in terms of the quality and execution time. They are also compared with 11 hash functions available in the literature using the real network data collected in our computer network.

The rest of the paper is organized as follows. Section 2 briefly surveys the principles of hash functions, LGP and evolutionary design of hash functions. The proposed approach to the evolutionary design of hash functions using LGP is introduced in Section 3. Section 4 presents the obtained results in terms of properties of evolved hash functions, their quality and execution time. Conclusions are given is Section 5.

2. RELATED WORK

This section covers relevant research conducted in the area of hash function design and evolutionary design using LGP.

2.1 Hash functions

A *hash function* is a mathematical function h that maps an input binary string (of length D) to a binary string of fixed length (R), $h : 2^D \rightarrow 2^R$, where $D >> R$. The output value is called hash value or simply hash [17].

Hash functions have many applications, for example, hash tables, caches and cryptography primitives employ them. Hash functions are primarily used in hash tables to quickly locate a data record if its search key is given. The hash function is then used to map the search key to an index which gives the place in the hash table where the corresponding record is located.

The quality of the hash function primarily determines the access time to data and table load factor that can be achieved for a given memory size. An important requirement on hash functions is that a small change in the input should generate a large change in the output. This is called the avalanche effect. The definition of hash function implies the existence of collisions, i.e. $h(x) = h(y)$, where x, y are two input messages such that $x \neq y$. The optimization of hash functions usually involves both criteria – maximizing the avalanche effect and minimizing the collision rate.

There are two types of hash functions, cryptographic and non-cryptographic hash functions. Cryptographic hash functions are used in security applications. Their basic property is that they are considered practically impossible to invert, that is, to recreate the input data from their hash values

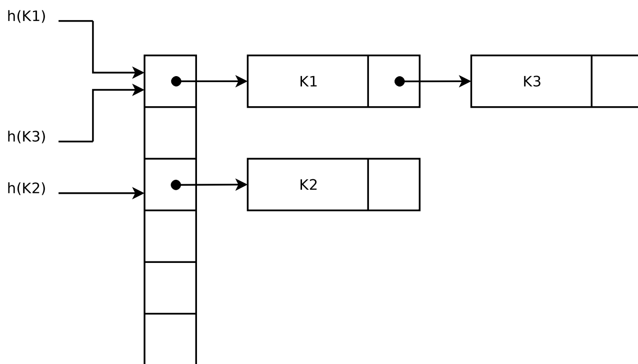

Figure 1: Hash table with separate chaining.

```
double LGP (double x ){
  r[0] = x

  r[2] = r[0] * r[0]
  r[1] = r[2] + r[0]
  r[3] = r[1] + r[0]
  r[3] = r[3] + r[2]
  r[0] = r[2] * r[1]
  r[1] = r[1] + r[4]
  r[0] = r[0] + r[3]
  r[0] = r[1] * r[0]
  return r0
}
```

Figure 2: Example of LGP individual.

alone. Cryptographic hash functions have to fulfill additional requirements, for example, first preimage resistance and collision resistance [19]. These requirements lead to a more complicated construction procedure and the hash function needs more time to compute the hash value.

Non-cryptographic hash functions, which this paper deals with, are typically used for fast lookup in hash tables [17] and they are much easier to design [20]. Various approaches have been developed to handle the collisions. For example, a separate chaining method manages a list of records having the same hash, see Fig. 1. Each slot in the table refers to a linear list where the data are stored. The hash value is computed for a given key and the data are stored to the first empty slot in the list addressed by the hash. This method is widely used, because it needs only elementary data structures and simple operations on lists. Other methods resolving the collisions are, for example, open addressing, linear probing, and cuckoo hashing.

Many (non-cryptographic) hash functions have been proposed, for example, DJBHash [4], DEKHash [17], FVN (Fowler-Noll-Vo) [12], One At Time and Lookup3 [13]. MurmurHash2 and MurmurHash3, which are utilized in many open source projects, are hash functions suitable for general hash-based lookup [1]. CityHash is a family of non-cryptographic hash functions designed for fast hashing of strings [23]. For hashing of the network flows, the so-called XOR folding has been proposed [6].

2.2 Linear Genetic Programming

Linear genetic programming [5, 21, 27] uses a linear representation of computer programs. Every program is composed of operations called instructions and operands stored in registers. Example of a candidate program is given in Figure 2. There are essentially two types of linear GP: machine code GP, where each instruction is directly executable by the CPU, and interpreted linear GP, where each instruction is executable by a virtual machine (simulator) implemented for a given processor.

An instruction is typically represented by the instruction code, destination register and two source registers, for example, $[+, r0, r1, r2]$ is representing the operation $r0 = r1 + r2$. The input data are stored in predefined registers or in an external memory. The result is returned in a predefined register. The number of instructions in a candidate program is variable, but the minimal and maximal values are defined. The number of registers available in a register ma-

chine is constant. The function set known from GP corresponds with the set of available instructions. The instructions are general-purpose (e.g. addition and multiplication) or domain-specific (e.g. read sensor 1). Conditional and branch instructions are important for solving general problems. Protected versions of instructions (e.g. a division returning a predefined value even if the divisor is zero) are employed in order to execute all programs without invoking exceptions such as division by zero.

New candidate programs are created using standard genetic operators such as crossover and mutation operating over lists of instructions. Advanced genetic operators have been proposed for LGP, for example [7, 9].

The most computationally expensive part of LGP is the fitness function evaluation. In order to obtain program's quality, the candidate program is executed with a set of training inputs, program's outputs are collected and compared with desired values. In a multi-objective scenario, non-functional program parameters such as the number of instructions can be optimized together with the functionality. We will employ a specific approach, see Section 3.3.

An individual can contain unused code parts, called bloat, which do not affect the fitness value. However, the bloat slows down the program execution. If bloat is detected and deleted, the evaluation time can significantly be reduced.

Parallel implementations of EAs are very popular because it is not usually difficult to parallelize the EA and obtained speedup can be significant. Parallel processing can be introduced at different levels of LGP: a parallel evaluation of candidate solutions, a parallel evaluation of training vectors or a parallel search in separate subpopulations.

A parallel LGP based on the island model operates with several subpopulations (the so-called islands) in which the evolution is conducted separately, but occasional exchange of the genetic material is permitted. The communication between islands can be either synchronous or asynchronous. As the evaluation of population(s) on different islands may consume different time, the asynchronous approach enables the islands to exchange genetic material when it is ready, i.e. the faster islands do not have to wait for the slower islands as in the case of synchronous communication.

2.3 Evolution of hash functions

In order to evaluate a hash function, a data set has to be applied and its key characteristics such as the number of collisions and the output distribution have to be computed. The quality of hashing on a particular data set then serves as the fitness score.

In papers [11, 10], GP employed the avalanche effect as the fitness criterion. In another work, the number of collisions was the main optimization target [14]. Cryptographic hash functions were evolved by means of gene expression programming in [25]. Hash functions tailored for a hardware implementations were obtained in [26]. Recently, non-cryptographic hash functions based on linear and non-linear feedback shift registers were evolved with the aim of efficient hardware implementation in FPGAs. It was shown that evolved solutions can achieve better table load factor in comparison with human-created solutions [8]. Finally, cache mapping functions, which can be considered as special instances of hash functions were evolved to optimize parameters of processor cache for a particular application [15].

3. HASH FUNCTION DESIGN

The main goal of this paper is to evolve using LGP a special hash function for hashing of network flows by means of a hash table with separate chaining.

3.1 Towards fast hashing

Each network flow is uniquely identified in IPv4 by a 5-tuple (source IP address (32 b), destination IP address (32 b), source port (16 b), destination port (16 b) and transport protocol (8 b)). In SDM, the network flow identifier has a constant length of 104 bits. As the target hash function has to accept only the 104 bit input, there is an opportunity to create a simple specialized hash function with good parameters. Universal hash functions consume the input data 'block by block' and the blocks are sequentially processed in a loop. Restricting the input to 104 bits enables to process the whole input string in one step, without any loops, which would significantly contribute to our key objective—shortening the execution time.

The second factor influencing the execution time is the instruction (function) set. Universal hash functions typically contain instructions such as logical XOR, addition, multiplication and rotation. The most computationally expensive operation is multiplication. Hence our objective will be to evolve multiplication less hash functions.

Finally, the number of instructions to be executed influences the execution time. After many experiments with LGP, we learned that sufficiently good hash functions require less than 12 instructions. Rather than applying a multiobjective LGP searching for a good compromise between the execution time and quality of hash functions, we propose to use a single-objective LGP in which the goal is to maximize the quality of hashing assuming that the program size is restricted. The validity of this approach is discussed in Section 4.3.

3.2 Parallel LGP and its parameters

The proposed implementation utilizes the island-based asynchronous parallel LGP model with a ring topology. After a predefined number of generations, every island sends the best individuals to its neighbors. All islands try to receive new individuals from other islands in every generation. Newly incoming individuals replace randomly chosen individuals of the population. However, the best individual of a given subpopulation is never replaced. The individuals are sent as integer array messages. In our case, the implementation is based on MPI [18]. LGP is employed in the style of [5].

The program size is restricted to contain up to 12 instructions. The set of constants consists of prime numbers that are commonly used in cryptographic hash function SHA-2 [2]. The function set includes the addition, logical XOR and right rotation. Note that right rotation and left rotation are interchangeable [11]. All LGP parameters are summarized in Table 1. They were chosen carefully on the basis of many experiments. The impact of some of them on the process of evolution will be discussed in Section 4.

3.3 Initialization and fitness function

The initial population is randomly generated. In order to calculate the fitness score, the responses have to be calculated for all training vectors. In this process, every training vector is used to initialize the registers of a candidate hash

Table 1: LGP parameters

Parameter	Value
Population size	200
Crossover probability	90 %
Mutation probability	15 %
Program length	12
Registers count/type	8/32 b – int
Constants	{0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a, 0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19, 0x428a2f98, 0x71374491}
Instruction set	{RightRotation, XOR, +}
Tournament size	4
Maximum number of generations	1000
Crossover type	One-point
Migration period	40 generations

function. All registers are 32 bit. The dimension of a training vector is reduced before starting the evolution to 3 x 32 bits in such a way that the source and destination IP addresses remain in the original format and a new 32 bit vector is created from the source and destination port (sp, dp) and transport protocol (tp) according to formula

$$((sp << 16) \vee dp) \oplus tp.$$

As real traffic contains especially two types of transport protocol (TCP and UDP) there is not a significant loss of information using this reduction of input vector. As this modification reduces the input space, it makes the hash function evolution easier.

The fitness function is based on counting the number of collisions. Let K_i inputs (keys) be mapped into i-th memory slot by a candidate hash function h. Then the fitness $f(h)$ is defined as

$$f(h) = \sum_{i=1}^{s} g_i, \text{ where} \quad (1)$$

$$g_i = \begin{cases} 0 & \text{if } K_i \leq 1 \\ \sum_{j=2}^{K_i} j^2 & \text{if } K_i \geq 2 \end{cases} \quad (2)$$

and s is the number of memory slots. This function penalizes candidate individuals showing many collisions and long lists in the hash table with separate chaining. Shorter lists in the table will lead to faster lookup. Lower fitness values mean better solutions. Example: Consider that two inputs are assigned to slot $i = 5$, three inputs are assigned to slot $i = 12$ and 0 or 1 inputs are assigned to the remaining slots. Then $f(h) = 2^2 + (2^2 + 3^2) = 17$.

4. EXPERIMENTS AND RESULTS

This section introduces the network data used for the evaluation and a set of hash functions that will be compared with evolved hash functions. The experimental evaluation is focused on a basic statistical evaluation of LGP. Then, the quality and time of execution of evolved non-cryptographic hash functions intended for a hash table with separate chaining are analyzed.

4.1 Network Data

Experiments will be performed with three data sets containing 20,000 (DataSet1), 50,000 (DataSet2) and 100,000 (DataSet3) identifiers of network flows. These sets were collected using a network monitoring device installed in our computer network in different days and are considered as the representative data for our network. There are no duplicate records in these data sets. *DataSet1* is used as a *training set* for LGP. IP addresses and transport protocol are converted to the decimal format which is used in our data sets (Figure 3).

4.2 Hash functions used for comparison

Evolved hash functions will be compared with human-created hash function DJBHash, DEKHash, One At Time, Lookup3, FVNHash, Murmur2, Murmur3, CityHash, a special hash function XORHash optimized for network flows [6] and evolved hash functions available in the literature GPHash [10, 11] and EFHash [14]. A 16 bit hash table with separate chaining is employed for testing all functions. As conventional hash functions typically produce a 32-bit hash value, we created a 16-bit output using XOR folding [6].

4.3 Analysis of LGP Setting

The evolution has been carried out using 1, 2, 4, 8 and 16 independent islands (i.e. cores) on a 16-core processor enabling the parallel processing and communication using MPI.

In order to obtain basic statistics, 20 independent LGP runs were performed, each taking 1000 generations (on each island). In other words, the total time allocated for the evolution is almost identical independently of the number of islands, but the number of generated individuals is linearly depending on the number of islands. The objective is to investigate how the quality of results is depending on available cores. The progress of evolution can be seen as the median value (out of 20 runs) in Figure 4. While the individuals were significantly improving for 100 generations, only small improvements are visible after 200 generations. Hence enabling 1000 generations for these experiments was more than sufficient.

The boxplots shown in Fig. 5 give the fitness value after 1000 generations spent by LGP executed with a different number of islands. Boxplots used in this figure represent the minimum, first quartile, median, third quartile and maximum. The experiments confirmed our assumption that if more islands are involved a better solution can be obtained, because more individuals are generated (in total) and exchanged among the islands. It has to be emphasized that we are not interested in an analysis of the speedup obtained by a parallel implementation in this case.

Fig. 6 shows the number of instructions that were really

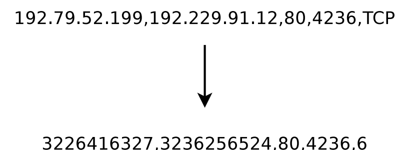

Figure 3: Example of conversion between a real network record and training vector.

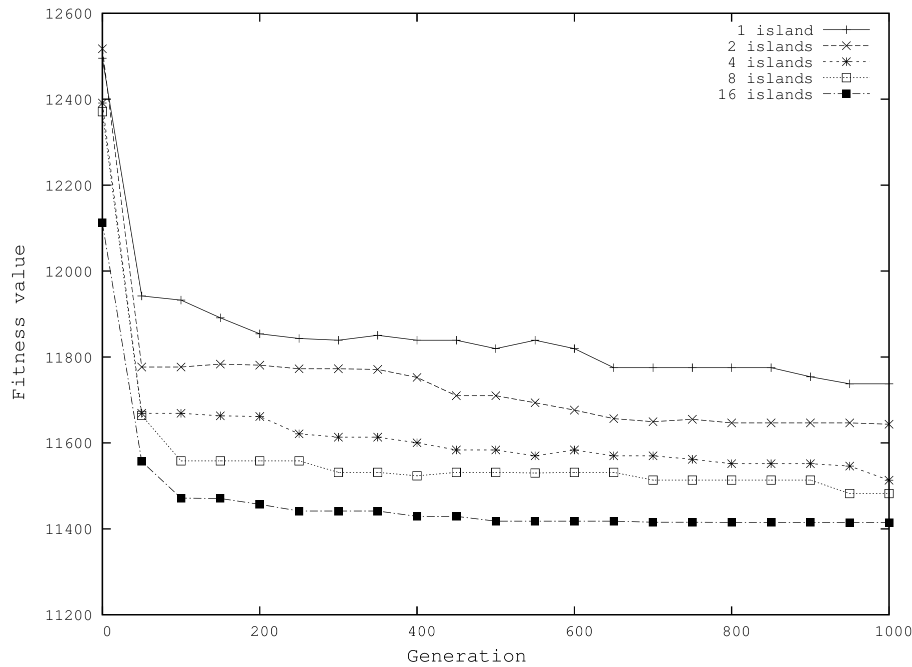

Figure 4: The progress of the best fitness score as median out of 20 independent runs on a different number of islands.

utilized in the programs created randomly for the initial population and in the programs of the final population. Please note that the instructions which did not contribute to the fitness (i.e. bloat) were removed. Even if the maximum program size is limited to 20 instructions, the median number of used instructions is less than 12. This analysis justifies our initial choice to limit the number of instructions to 12.

4.4 Evolved hash functions

From evolved solutions, two interesting hash functions were chosen for a detailed analysis. LGPHash1 (see the C code in Fig. 7) is the best scored hash function from all the runs. The second hash function selected is LGPHash2 (see the C code in Fig. 8) which is very simple. It ranked in the first quartile for 16 islands. It has to be noted that we removed all instructions not contributing to the fitness from evolved genotypes before creating the source codes in C which are presented in the paper.

In order to evaluate the impact of multiplication in the instruction set and the impact of increasing the number of instructions, we repeated our experiments (i) with a modified function set in which the multiplication was permitted and (ii) with up to 20 instructions allowed in the hash function.

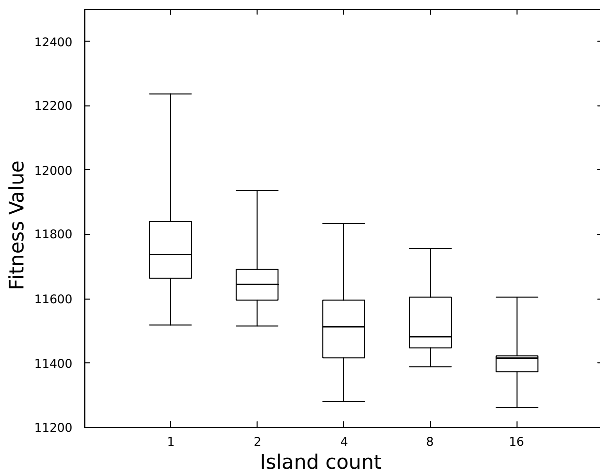

Figure 5: The best fitness values obtained from 20 independent runs on 1, 2, 4 , 8 and 16 islands.

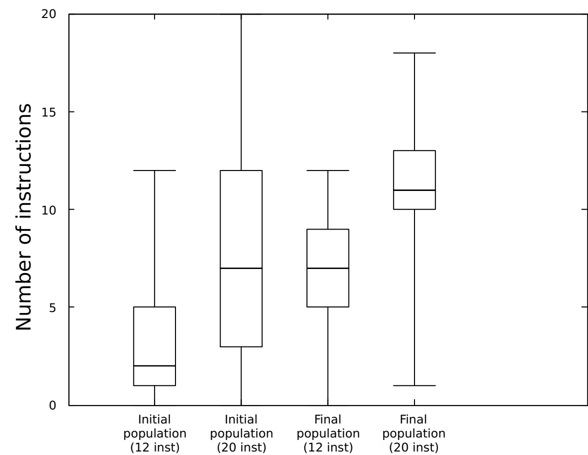

Figure 6: The number of instructions that were utilized in the initial population and final population if the program size is limited to 12 and 20 instructions.

```
unsigned int LGPHash1 (unsigned int * input ){
  r[0] = input[0]
  r[1] = input[1]
  r[2] = input[2]

  r[1] = r[1] + r[2]
  r[2] = r[1] + r[2]
  r[4] = r[0] + r[2]
  r[0] = r[1] + r[4]
  r[3] = 0x5BE0CD19
  r[2] = rotr(r[3], r[4])
  r[0] = r[0] + r[2]
  r[0] = 0xA54FF53A + r[0]
  return r0 ⊕ (r0 >> 16)
}
```

Figure 7: Evolved hash function LGPHash1.

```
unsigned int LGPHash2 (unsigned int * input ){
  r[0] = input[0]
  r[1] = input[1]
  r[2] = input[2]

  r[0] = r[0] ⊕ r[1]
  r[0] = r[2] + r[0]
  return r0 ⊕ (r0 >> 16)
}
```

Figure 8: Evolved hash function LGPHash2.

Evolved hash functions showing the best fitness value out of all runs—LGPhashMult (Fig. 9) for (i) and LGPhash20inst (Fig. 10) for (ii)—will be reported for comparison.

4.5 Collision test

Evolved hash functions as well as the hash functions obtained from the literature have been implemented in C programming language and compiled with the identical compiler setting. All tests were then preformed using these implementations.

Table 2 gives the number of collisions for all hash functions on three data sets. The best values are typed with bold font. It can be seen that the number of collisions is very similar for all the hash functions except *EFHash*. It can be concluded that evolved hash functions that are composed of simple instructions exhibit the quality almost identical with other hash functions. Neither enabling multiplication (LGPhashMult) nor more instructions (LGPhash20inst) have led to a considerable reduction in the number of collisions.

```
unsigned int LGPhashMult (unsigned int * input ){
  r[0] = input[0]
  r[1] = input[1]
  r[2] = input[2]

  r[6] = r[2] + r[0]
  r[7] = 0xA54FF53A
  r[5] = rotr(r[1], r[6])
  r[6] = r[1] ⊕ r[5]
  r[4] = r[6] * r[0]
  r[7] = rotr(r[1], r[7])
  r[6] = rotr(r[7], r[4])
  r[3] = r[6] + r[2]
  r[0] = r[3] + r[0]
  return r0 ⊕ (r0 >> 16)
}
```

Figure 9: Evolved hash function LGPhashMult.

```
unsigned int LGPhash20inst (unsigned int * input ){
  r[0] = input[0]
  r[1] = input[1]
  r[2] = input[2]

  r[6] = rotr(r[1], r[2])
  r[1] = r[1] ⊕ r[0]
  r[7] = r[1] + r[4]
  r[7] = r[7] + r[6]
  r[1] = rotr(r[7], r[6])
  r[0] = r[4] + r[6]
  r[5] = r[1] + r[0]
  r[7] = r[5] + r[2]
  r[4] = rotr(r[1], r[1])
  r[4] = r[7] ⊕ r[4]
  r[0] = r[0] ⊕ r[4]
  return r0 ⊕ (r0 >> 16)
}
```

Figure 10: Evolved hash function LGPhash20inst.

Table 2: The number of collisions.

Hash function	The number of collisions		
	DataSet1	DataSet2	DataSet3
DJBHash	2835	15113	48925
DEKHash	2926	15247	49017
FVNHash	2756	14957	48780
One At Time	2821	14988	**48636**
lookup3	2742	15009	48737
Murmur2	2800	15050	48749
Murmur3	2744	**14911**	48763
CityHash	2807	14990	48647
XORHash	2864	15011	48575
GPHash	2777	15052	48750
EFHash	5317	25266	63175
LGPhash1	***2667***	*15031*	*48680*
LGPhash2	*2746*	*15170*	*48835*
LGPhashMult	2769	14975	48715
LGPhash20inst	2761	14980	48755

4.6 The execution time

The execution time of hash functions (i.e. their implementations in C) was measured on the Intel XEON E5-2630 processor. Table 3 gives the average execution time obtained from 20 independent runs for all vectors of a given data set. Differences between the run times on the same data sets are very small which can be documented on detailed boxplots depicted in Fig. 11, where we compared the best evolved hash functions and the fastest conventional function XORHash.

The proposed special construction of loop-less and multiplication-less hash functions produced the faster solution. Enabling the multiplication definitely increases the execution time, but as the number of instructions is limited to length 12, evolved hash function containing the multiplication is

Table 3: The average execution time.

Hash function	Time [ms]		
	DataSet1	DataSet2	DataSet3
DJBHash	1.783	5.036	13.254
DEKHash	1.592	4.591	12.199
FVNHash	1.678	4.647	12.373
One At Time	2.365	6.269	15.763
lookup3	1.275	3.736	9.931
Murmur2	1.314	3.820	10.153
Murmur3	1.590	4.434	11.568
CityHash	3.089	7.883	19.237
XORHash	0.913	3.174	8.708
GPHash	1.936	6.229	15.813
EFHash	2.323	16.282	56.921
LGPhash1	*0.818*	*3.039*	*8.446*
LGPhash2	***0.756***	***2.852***	***8.057***
LGPhashMult	0.912	3.349	9.096
LGPhash20inst	0.916	3.242	8.954

Table 4: Overall quality of hash functions

Hash function	Quality (Q)		
	DataSet1	DataSet2	DataSet3
DJBHash	1.005	1.004	1.006
DEKHash	1.012	1.012	1.012
FVNHash	0.999	0.998	1.001
One At Time	1.003	1.001	**1.000**
lookup3	0.999	**1.000**	0.999
Murmur2	1.001	1.001	**1.000**
Murmur3	0.999	0.998	1.001
CityHash	1.003	0.999	0.998
XORHash	1.007	0.999	0.997
GPHash	1.001	1.003	**1.000**
EFHash	1.338	4.045	6.312
LGPhash1	*0.996*	*1.002*	*0.999*
LGPhash2	*0.999*	*1.003*	*1.001*
LGPhashMult	**1.000**	0.998	**1.000**
LGPhash20inst	0.998	0.998	**1.000**

still faster than other hash functions. If 20 instructions can be used, the execution time is prolonged proportionally to the number of instructions in the candidate program.

4.7 Overall quality of hash functions

The Compilers, Principles, Techniques book [3] proposes the following formula for evaluating the hash function quality:

$$Q = \sum_{j=0}^{m-1} \frac{b_j(b_j+1)/2}{(n/2m)(n+2m-1)}, \tag{3}$$

where b_j is the number of items assigned to j-th slot, m is the number of slots, and n is the total number of items. The numerator estimates the number of slots a hash function should visit to find the required value. The denominator is the number of visited slots for an ideal function that puts each item into a random slot. An ideal function produces the outputs with almost random distribution probability. If the hash function is ideal the formula should return 1, and a good quality is between 0.95 and 1.05. If Q is greater than 1, there are more collisions. If the number is smaller, there are less collisions than randomly distributing function.

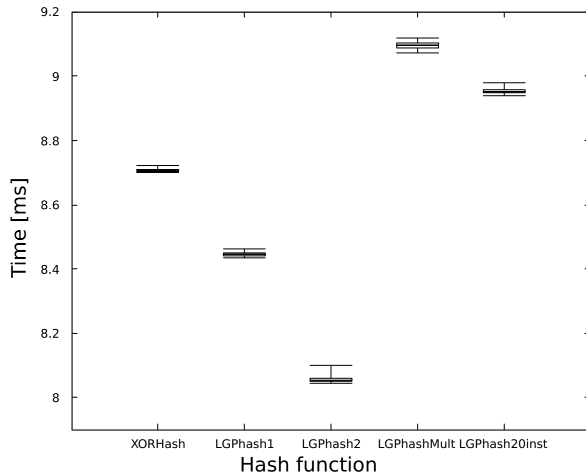

Figure 11: The execution time of selected hash functions on DataSet3 calculated from 20 runs.

From Table 4 it can be seen that evolved hash functions, despite the fact that they are composed of simple instructions, show very good quality according to the Q function [3]. This measurement indicated that enabling the multiplication and more instructions in programs has only a very small impact on the quality of hashing.

5. CONCLUSIONS

A method based on LGP was proposed which is capable of evolving high-quality and fast hash functions intended for network applications. In order to evolve desired hash functions, the function set was composed of simple instructions and the program size was restricted to 12 instructions. The fitness function was based on counting the number of collisions and penalizing candidate hash functions generating many collisions.

The best evolved hash functions were compared with 11 hash functions available in the literature. In order to provide a fair comparison, all hash functions were implemented in C, compiled for the same processor and executed several times to obtain the average execution time and quality.

In terms of the execution time, the best evolved hash function LGPhash1 provides 10.4%, 4.2% and 3.0% improvement on DataSets 1, 2 and 3 against the fastest available hash function XORHash [6] while the number of collisions was reduced by 6.8% for DataSet1 and slightly increased by 0.1% and 0.2% for DataSets 2 and 3. LGPhash1 and XORHash perform almost identically according to the Q quality function. The obtained speedup seems to be small, but one has to consider that the hash function is called many times and total savings are very valuable. Moreover, LGPhash1 reduced the execution time by 48.5%, 31.4% and 26.9% for DataSets 1, 2 and 3 with respect to Murmur3 hash function, which is typically used in SDM and which, on the other hand, provides a slightly lower number of collisions.

We observed that by enabling the multiplication or by increasing the program size, the number of collisions can be improved only insignificantly, but the execution time increased by 5-10%.

In our future work, we plan to analyze the impact of

pipeline processing and instruction scheduling which could influence the execution time on a particular processor. We will also test evolved hash functions in a SDM system.

Acknowledgment

This work was supported by the Czech science foundation project 14-04197S.

6. REFERENCES

[1] Murmur hash functions. https://github.com/aappleby/smhasher.

[2] Secure hashing. http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques*. Addison Wesley, 1986.

[4] D. J. Bernstein. Mathematics and computer science. https://cr.yp.to/djb.html.

[5] M. Brameier and W. Banzhaf. *Linear genetic programming*. Springer, New York, 2007.

[6] Z. Cao and Z. Wang. Flow identification for supporting per-flow queueing. In *Proc. of the Ninth International Conference on Computer Communications and Networks*, pages 88–93. IEEE, 2000.

[7] M. Defoin Platel, M. Clergue, and P. Collard. Maximum homologous crossover for linear genetic programming. In *Genetic Programming*, volume 2610 of *Lecture Notes in Computer Science*, pages 194–203. Springer Berlin Heidelberg, 2003.

[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*, pages 1214–1219. IEEE, 2015.

[9] C. Downey, M. Zhang, and W. N. Browne. New crossover operators in linear genetic programming for multiclass object classification. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 885–892. ACM, 2010.

[10] C. Estebanez, J. C. Hernandez-Castro, A. Ribagorda, and P. Isasi. Evolving hash functions by means of genetic programming. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1861–1862. ACM, 2006.

[11] C. Estébanez, J. C. Hernández-Castro, A. Ribagorda, and P. Isasi. Finding state-of-the-art non-cryptographic hashes with genetic programming. In *Parallel Problem Solving from Nature-PPSN IX*, pages 818–827. Springer, 2006.

[12] G. Fowler, P. Vo, and L. C. Noll. FVN Hash. http://www.isthe.com/chongo/tech/comp/fnv/.

[13] B. Jenkins. A hash function for hash table lookup. http://www.burtleburtle.net/bob/hash/doobs.html.

[14] J. Karasek, R. Burget, and O. Morsky. Towards an automatic design of non-cryptographic hash function. In *34th International Conference on Telecommunications and Signal Processing (TSP)*, pages 19–23. IEEE, 2011.

[15] P. Kaufmann, C. Plessl, and M. Platzner. EvoCaches: Application-specific Adaptation of Cache Mappings. In *Adaptive Hardware and Systems (AHS)*, pages 11–18. IEEE CS, 2009.

[16] L. Kekely, J. Kucera, V. Pus, J. Korenek, and A. Vasilakos. Software defined monitoring of application protocols. *IEEE Transactions on Computers*, 65(2):615–626, 2016.

[17] D. E. Knuth. The art of computer programming (volume 3). 1973.

[18] E. Lusk, S. Huss, B. Saphir, and M. Snir. MPI: A message-passing interface standard, 2009.

[19] W. Mao. *Modern cryptography: theory and practice*. Prentice Hall Professional Technical Reference, 2003.

[20] W. D. Maurer and T. G. Lewis. Hash table methods. *ACM Computing Surveys (CSUR)*, 7(1):5–19, 1975.

[21] M. Oltean and C. Grosan. A comparison of several linear genetic programming techniques. *Complex Systems*, 14(4):285–314, 2003.

[22] R. Pagh and F. F. Rodler. Cuckoo hashing. In *Algorithms - ESA 2001*, LNCS 2161, pages 121–133. Springer, 2001.

[23] G. Pike and J. Alakuijala. Introducing cityhash, 2011.

[24] A. Tongaonkar, R. Keralapura, and A. Nucci. Challenges in network application identification. In *Presented as part of the 5th USENIX Workshop on Large-Scale Exploits and Emergent Threats*, Berkeley, CA, 2012. USENIX.

[25] S. Varrette, J. Muszynski, and P. Bouvry. Hash function generation by means of gene expression programming. *Annales UMCS, Informatica*, 12(3):37–53, 2013.

[26] H. Widiger, R. Salomon, and D. Timmermann. Packet classification with evolvable hardware hash functions - an intrinsic approach. In *Second International Workshop on Biologically Inspired Approaches to Advanced Information Technology, BioADIT 2006*, pages 64–79, 2006.

[27] G. Wilson and W. Banzhaf. A comparison of cartesian genetic programming and linear genetic programming. In *Genetic Programming*, volume 4971 of *Lecture Notes in Computer Science*, pages 182–193. Springer, 2008.