Multi-objective Evolution of Hash Functions for High Speed Networks

David Grochol and Lukas Sekanina
Brno University of Technology, Faculty of Information Technology, IT4Innovations Centre of Excellence
Brno, Czech Republic
Email: igrochol@fit.vutbr.cz, sekanina@fit.vutbr.cz

***Abstract*—Hashing is a critical function in capturing and analysis of network flows as its quality and execution time influences the maximum throughput of network monitoring devices. In this paper, we propose a multi-objective linear genetic programming approach to evolve fast and high-quality hash functions for common processors. The search algorithm simultaneously optimizes the quality of hashing and the execution time. As it is very time consuming to obtain the real execution time for a candidate solution on a particular processor, the execution time is estimated in the fitness function. In order to demonstrate the superiority of the proposed approach, evolved hash functions are compared with hash functions available in the literature using real-world network data.**

I. Introduction

Many hardware providers have announced support for 100 gigabit-per-second (Gb/s) networks to overcome current 10-40 Gb/s solutions. Commercial companies, data and supercomputer centers, and other entities around the world are now working towards launching 100 Gb/s networks in order to benefit from faster communication and wider bandwidth for high-throughput requesting applications such as high-performance computing or high-quality video streaming. Managing 100 Gb/s networks, however, requires more precise performance monitoring (involving bandwidth monitoring, traffic analytics and anomaly detection) than in the previous era.

In order to effectively monitor and analyze high speed networks at the level of packet contents, *software defined monitoring* (SDM) concept has been developed [1]. Having less than 7 ns to process one packed in a 100 Gb/s network, SDM performs the analysis using relatively simple (and so fast) hardware whose functionality (i.e. rules of operation) are defined in software. Unrecognized traffic is then processed by sophisticated algorithms in software. The analysis is performed at the level of *flows*, where one flow is defined by five parameters within a certain time period: source and destination IP address, source and destination port and transport protocol. A memory address (slot) where the data of a given flow are stored is computed with a suitable *hash function*.

In our previous work, we employed linear genetic programming (LGP) to evolve high-quality hash functions for the software part of SDM [2]. In a single-objective design scenario and using real-world network traffic data, we obtained hash functions comparable in terms of quality of hashing, but faster than the state of the art hash functions. The objective for LGP was to minimize the number of collisions a given candidate hash function produces. As the hash function is called very often, it has to be very fast. However, the execution time of hash functions was not optimized. We just imposed an indirect constraint on the execution time requesting that the genotype must contain fewer than 12 instructions. Only simple elementary instructions such as addition and logic operations were allowed in the chromosome to minimize the execution time.

The goal of this paper is to show that if the execution time of a candidate hash function is formulated as a design objective together with the quality of hashing and the evolutionary design is performed with a multi-objective LGP, even better hash functions than those reported in paper [2] can be obtained. We propose and analyze an approach capable of estimating the execution time of a candidate hash function in the fitness function. The total execution time is estimated as the number of utilized instructions, where different weights are assigned to different types of instructions to reflect their different complexity. Scheduling and parallel execution of instructions on modern pipelined processors are also considered.

The estimated execution time and the number of collisions are then used as fitness functions in a multi-objective design algorithm based on LGP and NSGA-II. Evolved hash functions from the final Pareto front are compared with 11 hash functions available in the literature and 2 hash functions evolved in [2] using real-world network data.

The rest of the paper is organized as follows. Section II introduces the concept of hashing and hash function design. LGP and its utilization for hash function design in our previous approach is presented in Section III. Drawbacks of the previous approach are analyzed in Section III-C. The proposed multi-objective method is introduced in Section IV. Section V summarizes the experiments performed in order to evaluate the proposed method and compare resulting hash functions with existing solutions. Conclusions are given in Section VI.

II. Hash function design

This section surveys the principles of hash function design and their utilization in SDM. As this paper is devoted to software implementations of hash functions on common processors, circuit implementations of hash functions created for hardware parts of SDM (such as [3]) will not further be discussed. Moreover, we will not consider cryptographic hash

978-1-5090-4601-0/17/$31.00

functions that have to exhibit additional properties [4]. They are thus irrelevant for SDM.

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 [5].

The main purpose of hash functions is to locate (in constant time) a data record for a given search key, avoiding thus a sequential or log-time search in data records [5]. The quality of hash function is given in terms of the access time to data and table load factor (for a given memory size). 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 $x \neq y$. Good hash functions generate a big change in the output for a small change in the input. This is called the avalanche effect.

The hash function is typically called several times in order to obtain desired address because the memory addressing system can be designed as hierarchical, for example, in the cuckoo hashing scheme [6]. Hence, it is important to optimize not only the quality of hashing, but also the execution time, which is crucial for SDM as the hash function is called very often. Note that the worst case packet processing time is 7 ns for 100 Gb/s networks.

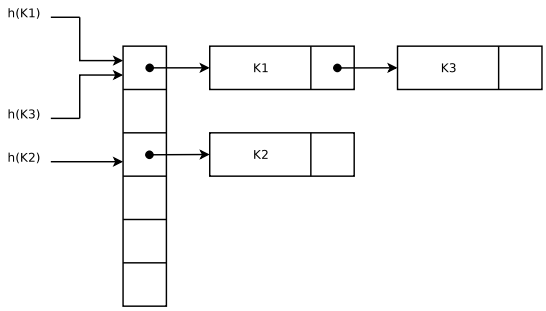

Fig. 1. Hash table with separate chaining.

Collisions introduced by a hash function can be managed in different ways in hash tables [7]. The most popular approach is a separate chaining method which operates a list of records having the same hash, see Fig. 1. Each slot in the table is pointing 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. The advantage is that the method requires only basic data structures and simple operations on lists.

The literature provides us with various implementations of hash functions including DJBHash [8], DEKHash [5], FVN (Fowler-Noll-Vo) [9], One At Time, Lookup3 [10], MurmurHash2, MurmurHash3 [11] and CityHash [12]. For hashing of the network flows, the so-called XOR folding has been proposed [13].

III. Previous work on evolution of hash functions

Genetic programming (GP) has been used to provide various hash functions. The fitness function reflecting the quality of hashing is usually based on measuring the avalanche effect [14], [15] or the number of collisions [16]. Cryptographic hash functions were designed with gene expression programming in [17]. Circuit-based hash functions were obtained in [18]. Hash functions are also employed in cache memories. An example of GP-based optimization of hash functions for particular applications is given in [19].

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

Fig. 2. Example of LGP individual.

This section briefly presents LGP and our previous single-objective LGP-based approach for the design of fast hash functions in SDM [2]. In particular, it analyzes weaknesses of the method that motivated the research presented in this paper.

A. Linear Genetic Programming

Linear genetic programming (LGP) [20], [21], [22] is a form of genetic programming in which candidate programs are encoded as sequences of instructions and executed on a register machine. Example of a candidate program is given in Figure 2.

In LGP, every instruction typically includes an operation (instruction code), one or two source registers and a destination register. One-register instructions operate with one register as the destination register (e.g. $r0$ = read sensor(); load constant to register $r1$ etc.). Two-register instructions operate with one source and one destination register (e.g. $r0 = sin(r1)$; $r0 = bitwise\_rotation(r1)$). Three-register instructions operate with two source registers and one destination register (e.g. $r0 = r1 + r2$). The number of instructions in a candidate program is variable, but the minimal and maximal values are usually defined. The number of registers available in a register machine is constant. The result is returned in a selected register. The function (instruction) set contains general-purpose (e.g. addition and multiplication) and domain-specific (e.g. read sensor) instructions. LGP is usually used with basic genetic operators (tournament selection, crossover, mutation). However, advanced genetic operators were also proposed, for example [23], [24].

B. LGP for Hash Function Design

In our previous work [2], LGP was used to deliver a special hash function for hashing of network flows by means of a hash table with separate chaining. Each network flow can be uniquely identified by a 5-tuple. For IPv4, the 5-tuple contains

source and destinations IP address (2×32 bits), source and destination ports (2×16 bits) and transport protocol (8 bits). As the network flow identifier has a constant length of 104 bits in SDM, the hash function evolved by LGP accepts only 104 bits. Restricting the input to 104 bits enabled to process the whole input string in one step, without sequential reading the input data and multiple executions of the hash function, shortening thus the execution time.

In order to even simplify the problem, the 104 bit input vector was reduced 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

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

No significant loss of information was reported after applying this simple approach.

LGP operated with a 32 bit register machine. Universal hash functions typically contain instructions such as logical XOR, addition, multiplication and rotation. Hence, we included these operations to our instruction set. Randomly created programs composed of these instructions constituted the initial population. We used standard genetic operators such as tournament selection, one-point crossover and mutation.

A single objective search was guided by the fitness function reflecting the quality of hashing. Let K_i inputs (keys) be mapped into i-th memory slot by a candidate hash function h. Then the fitness $f(h)$ was defined as the weighted number of collisions:

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

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

where s is the number of memory slots. This function clearly penalized candidate hash functions showing many collisions and thus long lists in the hash table with separate chaining. The objective was to minimize $f(h)$.

The execution time was controlled indirectly, by formulating a constraint that the maximum chromosome size is 12 instructions.

C. Lessons Learned

Experiments reported in [2] confirmed that LGP can evolve hash functions for SDM (i) that show at least the same quality of hashing as common hash functions and (ii) that are faster than these common functions. In order to perform a fair comparison with conventional hash functions that are available at the level of C code, evolved hash functions as well as 11 common hash functions were implemented in C, compiled (with the code optimization parameter -O3) for the same processor and executed many times to obtain the average execution time and quality on three data sets. One of the evolved hash functions, LGPhash1, reduced the execution time by 35% on average with respect to Murmur3 hash function [11], which is typically used in SDM. These results were obtained with the instruction set consisting of addition, XOR and shift operations. Enabling the multiplication operator in the instruction set improved the quality of hashing insignificantly, but the execution time increased by 5-10%. No improved was obtained by increasing the maximum chromosome size to 20 instructions.

Although we evolved good hash functions, we revealed the following drawbacks after detailed examination of the results: (1) As the chromosome could contain up to 12 instructions, we generated short and fast programs, but we did not optimize the execution time. Resulting hash functions were selected manually, on the basis of their functionality solely, i.e. we potentially overlooked faster hash functions showing good quality. Figure 3 reports the number of evolved hash functions (y-axis) with a particular execution time (x-axis) in a 200 member LGP population. The execution time is the average time from 20 independent runs of a particular hash function compiled for a target processor (Intel XEON E5-2620v3) and executed using a test set. The execution time of most hash functions is concentrated in the 1 ms - 2 ms interval, where we were looking for the best-performing hash functions for our comparisons. However, there exist much faster hash functions as seen around and below 1 ms on the x-axis.

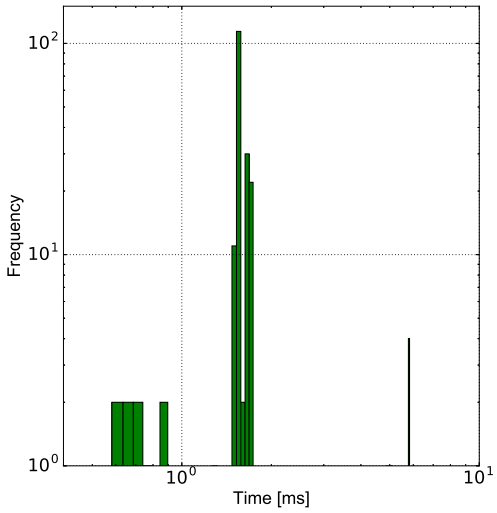

Fig. 3. The number of hash functions with a particular execution time in a 200 member LGP population.

(2) Counting the number of instructions in the fitness function can only indirectly reflect the execution time. The reason is that particular instructions have different execution times that have to be reflected in the correct estimate of the total execution time of the hash function. We measured the execution time of randomly generated 1 million instruction programs consisting of just one type of instructions and observed that multiplication is 3 times more expensive than other instructions used in hash functions. This observation is

consistent with clock cycles performed for given instructions by our processor.

(3) As modern processors introduce parallel processing at the level of instruction execution, real execution time depends on how the instructions are scheduled for parallel hardware pipelines. For example, if there are no dependencies between the instructions they can be executed in parallel, reducing thus significantly the total execution time of the hash function.

(4) The total execution time clearly depends on the quality of the hash function because fast but weak hash functions will generate many collisions and additional sequential processing of items in the hash table. Hence, a multi-objective optimization approach is needed.

IV. Multi-objective evolution of hash functions

In order to eliminate the drawbacks reported in the previous section and evolve hash functions showing good tradeoffs between the execution time and quality of hashing, we will construct the search algorithm as a multi-objective LGP minimizing two objectives: (i) the number of collisions (according to eq. 1) and (ii) the execution time. As it is very time consuming to obtain the real execution time for a candidate solution on a particular processor, the execution time will be estimated.

A. Execution Time Estimation

At the level of chromosome, the number of instructions can be restricted as recommended in [2]. However, a candidate fixed-size program can still contain unused code parts, called bloat, which do not affect the fitness value (result). There are two types of unused instructions. In the first case, there are instructions whose result is not used by any other instruction (structural redundancy). In the second case, there are instructions whose execution does not affect contents of registers (semantic redundancy). The proposed algorithm estimates the execution time as the number of instructions that will be executed when the candidate program is compiled and redundant instructions are removed. Note that multiplication is counted with weight 3, but this is omitted in the pseudo codes to keep them more readable. It is assumed that there is one register containing the output value.

Algorithm 1 removes structurally redundant instructions. In a candidate program p, last instruction i which modifies the output register is detected. Then, destination and source registers of instruction i are inserted to a set of used registers. In the next step, the algorithm moves backward in the candidate program and checks if a given instruction uses some registers from the set of used registers as the destination register. If so, source registers of such instruction are inserted to the set of used instructions. Every instruction affecting content of the output register thus increases the number of used instructions. Weights are assigned to some instructions to reflect their higher complexity.

Algorithm 1: Execution time estimation (simple)

Input: Candidate program p
Output: The number of used instructions

```
used-instructions = 0;
used-registers ← Insert(output-register);
while ⟨ i ← getLastInstruction(p) ⟩ do
    if DestinationRegister(i) ∈ used-registers then
        used-registers ← Insert(source-registers(i));
        Increment(used-instructions);
    remove instruction i from p;
return used-instructions;
```

Algorithm 2 performs a basic semantic analysis of a candidate program. It also captures the instruction level parallelism [25] known as SIMD (Single instruction multiple data). SIMD processing refers to a mechanism that enables to process multiple data with a single instruction. Modern CPUs can typically process 256 bits at once which means that eight 32-bit operations can be executed in one instruction instead of executing 8 instructions sequentially.

First, Algorithm 2 employs Algorithm 1 to remove structurally redundant instructions. In the next step, it is determined for all instructions when they can be executed. The ASAP (As Soon As Possible) routine checks if some source registers of

Algorithm 2: Execution time estimation (advanced)

Input: Candidate program p
Output: The number of used instructions

```
used-instructions = 0;
r ← remove structurally redundant instructions from p
    using Alg.1;
M ← create matrix for instructions;
for i in r do
    M ← using ASAP and ALAP routines to determine
        when i can be executed;
while ⟨ Some instruction(s) exist in M ⟩ do
    I ← find in M all instructions of the same type
        which can be executed together;
    remove I from M;
    increment(used-instructions);
return used-instructions;
```

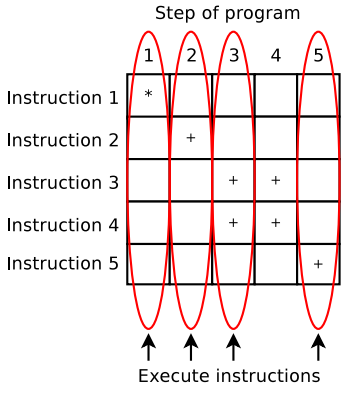

Fig. 4. Example of scheduling for a program given in Fig. 2. Instructions 3 and 4 can be executed together.

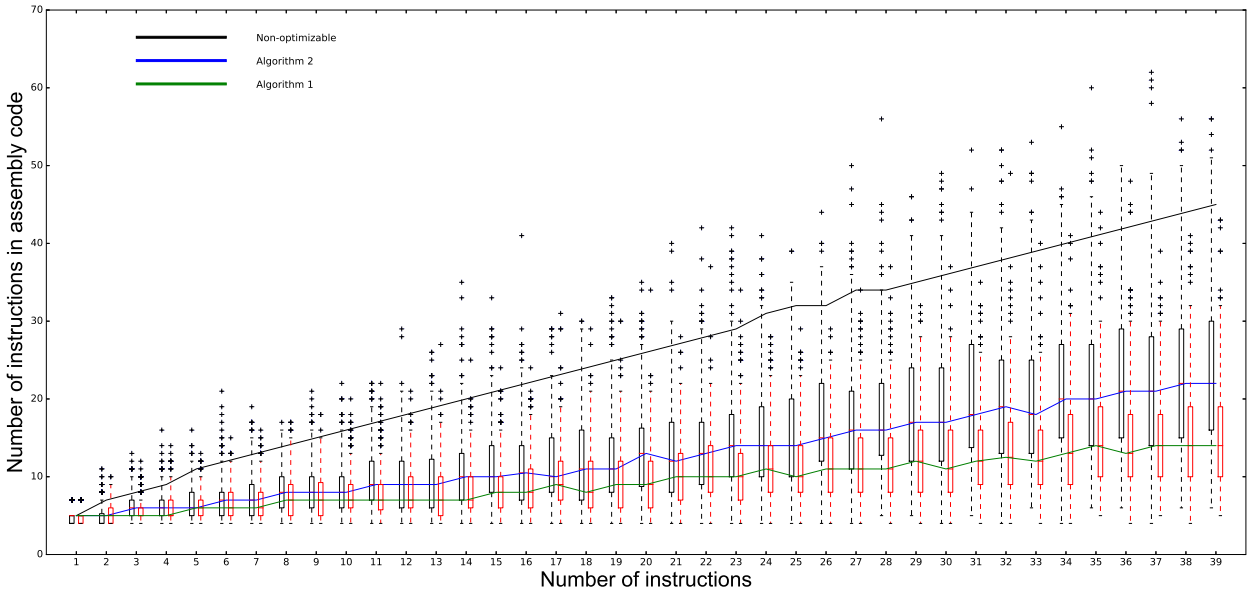

Fig. 5. The number of instructions in assembly code for 400 randomly generated programs containing 1 - 39 instructions, where the number of instructions was calculated according to Algorithm 1 (the median shown in green) and Algorithm 2 (the median is shown in blue). The size of non-optimizable programs is shown using the black line.

instruction i are modified by some previous instructions. These dependences are marked in matrix M. All instructions from the beginning up to this modification can be executed together with instruction i. If the destination register of instruction i is used in some previous instruction j as destination register, instruction j is deleted from M, because instruction i modifies the destination register without using its value. The ALAP (As Late As Possible) routine checks if the destination register of instruction i is used in some following instructions as a source register. If so, it is marked in M. If it is used as a destination register, instruction i is removed from M, because its value is not used. All instructions up to this modification can be executed together with instruction i. ASAP and ALAP identify all instructions that can be executed together.

In the next step, the algorithm identifies those instructions (of the same type) that can be executed together using one SIMD instruction on the CPU. It sequentially determines the largest overlaps of instructions, removes them from M and increases the number of used instructions. The routine is repeated until some instruction(s) exist in matrix M. Example of scheduling for the program given in Fig. 2 is shown in Fig. 4. In this case, only instructions 3 and 4 can be executed together. The last instruction has to be executed independently. The total number of instructions estimated by Algorithm 2 is 4. Algorithm 1 outputted 5 instructions (the weights reflecting the different complexity of instructions are not considered in our example).

In order to validate the proposed method, we compared the number of instructions produced by the C compiler for programs whose size was estimated by Algorithm 1 and Algorithm 2. We randomly generated 400 programs containing exactly k instructions according to Algorithm 1. We repeated the experiment, but the program size was assigned by Algorithm 2. The idea behind this experiment is that programs containing exactly k instructions according to Algorithm 1 have to be on average shorter in terms of assembly code generated by the C compiler than programs containing exactly k instructions according to Algorithm 2. The reason is that Algorithm 2 can eliminate semantic redundancy and parallel operations and hence "more instructions" are needed to reach k instructions in the random program generator. Fig. 5 compares Algorithm 1 and Algorithm 2 for $k = 1 \ldots 39$ instructions. Fig. 5 also contains the size of assembly code for manually created programs that are known to be non-optimizable by the compiler (black line). As the compiler adds some additional instructions, the assembly code size (y-axis) is slightly greater than estimated numbers (x-axis).

```
unsigned int NSGAHash1 (*input){
  r[0], r[1], r[2] = input;

  r[0] = r[0] + r[2];
  return r0 ⊕ (r0 >> 16);
}
```
NSGAHash1

```
unsigned int NSGAHash2 (*input){
  r[0], r[1], r[2] = input;

  r[4] = r[1] ⊕ r[0];
  r[0] = r[4] + r[2];
  return r0 ⊕ (r0 >> 16);
}
```
NSGAHash2

```
unsigned int NSGAHash3 (*input){
  r[0], r[1], r[2] = input;

  r[1] = rotr(r[0], 12);
  r[3] = r[4] + r[2];
  r[0] = r[1] + r[3];
  return r0 ⊕ (r0 >> 16);
}
```
NSGAHash3

```
unsigned int NSGAHash4 (*input){
  r[0], r[1], r[2] = input;

  r[1] = rotr(r[1], 22);
  r[6] = r[0] ⊕ r[6];
  r[3] = r[2] + r[6];
  r[0] = r[1] + r[3];
  return r0 ⊕ (r0 >> 16);
}
```
NSGAHash4

```
unsigned int NSGAHash5 (*input){
  r[0], r[1], r[2] = input;

  r[4] = r[1] ⊕ r[0];
  r[1] = rotr(r[4], 22);
  r[6] = r[0] + r[6];
  r[3] = r[2] + r[6];
  r[0] = r[1] + r[3];
  return r0 ⊕ (r0 >> 16);
}
```
NSGAHash5

```
unsigned int NSGAHash6 (*input){
  r[0], r[1], r[2] = input;

  r[7] = rotr(r[0], 7);
  r[4] = r[1] ⊕ r[0];
  r[1] = rotr(r[4], 22);
  r[6] = r[7] ⊕ r[6];
  r[3] = r[2] + r[6];
  r[0] = r[1] + r[3];
  return r0 ⊕ (r0 >> 16);
}
```
NSGAHash6

```
unsigned int NSGAHash7 (*input){
  r[0], r[1], r[2] = input;

  r[3] = rotr(r[2], 3);
  r[5] = rotr(r[1], 3);
  r[4] = r[0] * r[5];
  r[5] = rotr(r[4], 11);
  r[0] = r[5] ⊕ r[3];
  r[0] = r[4] + r[0];
  return r0 ⊕ (r0 >> 16);
}
```
NSGAHash7

Fig. 6. Evolved hash functions from the non-dominated set in Fig. 7.

TABLE I
LGP PARAMETERS

Parameter	Value
Population size	200
Crossover probability	90 %
Mutation probability	15 %
Program length	20
Registers count/type	8/32 b – int
Constants	{0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a, 0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19, 0x428a2f98, 0x71374491}
Instruction set (weight)	{RightRotation (1), XOR (1), + (1), * (3)}
Tournament size	4
Maximum number of generations	1000
Crossover type	One-point

B. LGP and NSGA-II

The proposed implementation is based on LGP as used in paper [2], but the search is conducted by means of NSGA-II [26]. The maximum program size is 20 instructions in order to provide more opportunities to find good tradeoffs. The function set includes typical instructions for hash function design (addition, multiplication, logical XOR and right rotation). The set of constants consists of the values that are used in the initial phase of cryptographic hash function SHA-2 [27].

The initial population is randomly generated. Two fitness functions are employed to measure (i) the collisions (according to eq. 1) and (ii) the execution time (according to Algorithm 2). All training vectors have to be evaluated to obtain the fitness score.

V. Experiments and Results

The experimental evaluation deals with evolved hash functions and their analysis in terms of quality of hashing and execution time. Results will be compared with conventional hash functions and hash functions evolved in [2].

A. Network Data

The network data used in our experiments were collected with a network monitoring device installed in our research computer network. Network data were divided into three data sets containing 20,000 (DataSet1), 50,000 (DataSet2) and 100,000 (DataSet3) identifiers of network flows. Note that the identifiers of network flows are unique. *DataSet1* is used as a *training set* for LGP.

B. Hash Functions Used for Comparison

The comparison is intended for the hash table with separate chaining. 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 [13], evolved hash functions available in the literature (GPHash [15], [14] and EFHash [16]) and the best hash functions LGPHash1 and LGPHash2 evolved for network flows by LGP in [2]. 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 [13].

C. Resulting Pareto fronts

In order to obtain the best setup of the algorithm, many independent runs with different parameters of the algorithm

were performed. Considering the obtained results and parameters given in paper [2], we used for final experiments the setting which is summarized in Tab. I. Note that all LGP runs reported in [2] stagnated after about 200 generations.

Fig. 7 shows Pareto fronts obtained from 30 independent runs of LGP. Results of one of the runs, which contains the best obtained solutions according to particular objectives (i.e. a solution showing minimum collisions and a solution showing the minimum number of instructions) were chosen for a detailed inspection. The corresponding Pareto front containing 7 unique hash functions is given in Fig. 7 (blue squares). For example, NSGAHash1 (see the C code in Fig. 6) is the hash function consisting of just one instruction. Its quality of hashing is not acceptable. On the other hand, NSGAHash7 (see the C code in Fig. 6) provides the best quality of hashing (in the selected run), but its execution time is the longest one.

D. The Number of Collisions

The hash functions obtained from literature and evolved hash functions were implemented in C programming language and compiled with the identical compiler settings. All tests were then performed with these implementations to ensure fair comparisons.

Table II gives the number of collisions for all hash functions on all data sets for 16 bit hash table. The best values are typed in bold. It can be seen that the multi-objective LGP provides hash functions with a very similar number of collisions as other hash functions, but there are solutions (NSGAHash6 and NSGAHash7) which excel over all available hash functions.

E. The Execution Time

Table III reports the average execution time obtained from 20 independent runs over all data sets. Note that hash functions having low number of instructions (such as NSGAHash1, NSGAHash2) do not show the shortest execution time. The reason is that the number of collisions produced by these hash functions is higher which means that more time is needed to accommodate incoming items in the table. NSGAHash4 provides the shortest execution time because a good tradeoff between the number of collisions and the complexity of the hash function was discovered. NSGAHash4 is even better than hash functions LGPhash1 and LGPhash2 discovered by means of a single-objective LGP in [2].

TABLE II
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
NSGAHash1	2923	15677	49336
NSGAHash2	2746	15170	48835
NSGAHash3	2689	15575	49292
NSGAHash4	2692	15010	48715
NSGAHash5	2759	14975	48749
NSGAHash6	2650	**14839**	48680
NSGAHash7	**2639**	14975	48650

TABLE III
THE AVERAGE EXECUTION TIME.

Hash function	Time $[ms]$		
	DataSet1	DataSet2	DataSet3
DJBHash	1.069	3.608	9.690
DEKHash	0.890	3.210	8.647
FVNHash	1.021	3.546	9.556
One At Time	1.361	4.568	12.024
lookup3	0.721	2.670	7.473
Murmur2	0.787	2.868	7.871
Murmur3	0.929	3.304	8.892
CityHash	0.760	2.736	7.603
XORHash	0.649	2.390	6.774
GPHash	1.448	4.749	12.406
EFHash	1.871	13.560	48.132
LGPhash1	0.591	2.913	6.588
LGPhash2	0.561	2.182	6.336
NSGAHash1	0.568	2.871	8.642
NSGAHash2	0.560	2.182	6.334
NSGAHash3	**0.541**	2.871	8.500
NSGAHash4	0.561	**2.168**	**6.267**
NSGAHash5	0.564	2.191	6.394
NSGAHash6	0.559	2.192	6.369
NSGAHash7	0.593	2.295	6.883

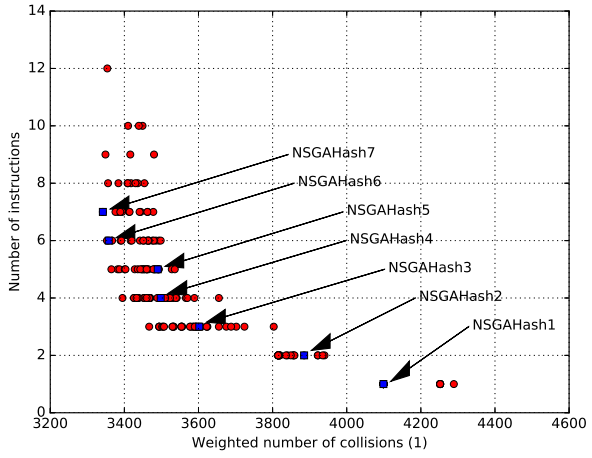

Fig. 7. Pareto fronts obtained from 30 independent runs. Selected hash functions (blue squares) are given in Fig. 6.

F. Overall Quality of Hash Functions

The quality of hashing can be expressed according to [28] as:

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

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

TABLE IV
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
NSGAHash1	1.010	1.476	1.566
NSGAHash2	0.999	1.003	1.001
NSGAHash3	0.996	1.470	1.560
NSGAHash4	0.996	0.999	1.998
NSGAHash5	0.998	0.998	**1.000**
NSGAHash6	0.992	0.995	0.999
NSGAHash7	0.993	0.999	1.001

numerator estimates the number of slots a hash function should visit to find the require 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 a nearly random distribution probability. If the hash function is ideal, the formula should return 1, a good quality is between 0.95 and 1.05.

According to this criterion, evolved hash functions as well as conventional hash functions were evaluated. The Q score follows the trend of the quality indicator used in LGP (the number of collisions) as we travel along the Pareto front.

VI. CONCLUSIONS

We proposed a multi-objective linear genetic programming approach to evolve fast and high-quality hash functions for common processors programmed as network flow monitoring devices. It was shown using real world network data that the proposed method provides better compromise solutions (in terms of execution time and quality of hashing) than commonly used hash functions and specialized hash functions evolved with a single-objective LGP. Our future work will be devoted to integrating the evolved hash functions to the SDM concept.

ACKNOWLEDGMENTS

This work was supported by the Czech science foundation project GP16-08565S.

REFERENCES

[1] L. Kekely, J. Kucera, V. Pus, J. Korenek, and A. Vasilakos, "Software defined monitoring of application protocols," *IEEE Transactions on Computers*, vol. 65, no. 2, pp. 615–626, 2016.

[2] D. Grochol and L. Sekanina, "Evolutionary design of fast high-quality hash functions for network applications," in *Proc. of the 2016 Genetic and Evolutionary Computation Conference*. ACM, 2016, pp. 901–908.

[3] R. Dobai, J. Korenek, and L. Sekanina, "Adaptive development of hash functions in fpga-based network routers," in *2016 IEEE Symposium Series on Computational Intelligence*. IEEE Computational Intelligence Society, 2016, pp. 1–8.

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

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

[6] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *Algorithms ESA 2001*, ser. LNCS 2161. Springer, 2001, pp. 121–133.

[7] W. D. Maurer and T. G. Lewis, "Hash table methods," *ACM Computing Surveys (CSUR)*, vol. 7, no. 1, pp. 5–19, 1975.

[8] D. J. Bernstein, "Mathematics and computer science," https://cr.yp.to/djb.html, [ONLINE, accessed: 31. 1. 2016].

[9] G. Fowler, P. Vo, and L. C. Noll, "FVN Hash," http://www.isthe.com/chongo/tech/comp/fnv/, [ONLINE, accessed: 31. 1. 2016].

[10] B. Jenkins, "A hash function for hash table lookup," http://www.burtleburtle.net/bob/hash/doobs.html, [ONLINE, accessed: 31. 1. 2016].

[11] "Murmur hash functions," https://github.com/aappleby/smhasher, [ONLINE, accessed: 31. 1. 2016].

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

[13] Z. Cao and Z. Wang, "Flow identification for supporting per-flow queueing," in *Computer Communications and Networks, 2000. Proceedings. Ninth International Conference on*. IEEE, 2000, pp. 88–93.

[14] 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*. Springer, 2006, pp. 818–827.

[15] 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*. ACM, 2006, pp. 1861–1862.

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

[17] S. Varrette, J. Muszynski, and P. Bouvry, "Hash function generation by means of gene expression programming," *Annales UMCS, Informatica*, vol. 12, no. 3, pp. 37–53, 2013.

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

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

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

[21] M. Oltean and C. Grosan, "A comparison of several linear genetic programming techniques," *Complex Systems*, vol. 14, no. 4, pp. 285–314, 2003.

[22] G. Wilson and W. Banzhaf, "A comparison of cartesian genetic programming and linear genetic programming," in *Genetic Programming*. Springer, 2008, pp. 182–193.

[23] M. Defoin Platel, M. Clergue, and P. Collard, "Maximum homologous crossover for linear genetic programming," in *Genetic Programming*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, vol. 2610, pp. 194–203.

[24] 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*. ACM, 2010, pp. 885–892.

[25] D. W. Wall, *Limits of instruction-level parallelism*. ACM, 1991, vol. 19, no. 2.

[26] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii," in *International Conference on Parallel Problem Solving From Nature*. Springer, 2000, pp. 849–858.

[27] "Secure hashing," http://csrc.nist.gov/groups/ST/toolkit/secure_hashing-.html, [ONLINE, accessed: 31. 1. 2016].

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