Multi-objective Evolution of Ultra-Fast General-Purpose Hash Functions

David Grochol(✉) and Lukas Sekanina

IT4Innovations Centre of Excellence, Faculty of Information Technology, Brno University of Technology, Božetěchova 2, 612 66 Brno, Czech Republic
{igrochol,sekanina}@fit.vutbr.cz

Abstract. Hashing is an important function in many applications such as hash tables, caches and Bloom filters. In past, genetic programming was applied to evolve application-specific as well as general-purpose hash functions, where the main design target was the quality of hashing. As hash functions are frequently called in various time-critical applications, it is important to optimize their implementation with respect to the execution time. In this paper, linear genetic programming is combined with NSGA-II algorithm in order to obtain general-purpose, ultra-fast and high-quality hash functions. Evolved hash functions show highly competitive quality of hashing, but significantly reduced execution time in comparison with the state of the art hash functions available in literature.

1 Introduction

Hash functions are highly nonlinear functions assigning a relatively short numerical representation to an arbitrary data record of a predefined structure and size. Hash functions are frequently used in many applications of computer science and engineering such as hash tables, caches and Bloom filters. Hash functions are evaluated with respect to two fundamental properties: (i) quality of hashing – which can be defined in different ways (see Sect. 2.1) and (ii) complexity, which is highly correlated with the execution time. Some additional properties are crucial for the so-called cryptographic hash functions, but this paper only deals with *non-cryptographic* hash functions. As the design of a good hash function is tricky and requires a lot of insight and experience, evolutionary algorithms (genetic programming (GP) in particular) have been employed to accomplish this task.

The existing body of literature dealing with evolutionary design of hash functions is relatively rich; however, except paper [1] none of them is explicitly oriented to the optimization of the time of execution (latency or delay in other words) which becomes crucial in contemporary high end applications such as high speed network monitoring, big data indexing and finding duplicate records.

In the literature, the latency is usually considered as a constraint and the optimization goal is to maximize the quality of hashing. The hash function design problem is then formulated as a single objective design problem.

M. Castelli et al. (Eds.): EuroGP 2018, LNCS 10781, pp. 187–202, 2018.
https://doi.org/10.1007/978-3-319-77553-1_12

In some cases, hash functions are evolved as application-specific functions and evaluated in a very specific environment [1–4], providing thus much better solutions in particular applications than the so called *general-purpose hash functions*. For example, a multi-objective evolutionary design approach focusing not only on the quality of hashing, but also on the execution time has been proposed for network flow hashing [1]. In this case, evolved hash functions had a fixed-size input (96 bits) and consisted of a linear sequence of instructions which is executed just once to obtain the hash.

The goal of this paper is to present and evaluate a *multi-objective evolutionary approach* for the design of high-quality and ultra-fast *general-purpose* hash functions. The main difference with respect to [1] is that the resulting hash functions are capable of accepting multiple k-bit inputs (in order to be general-purpose ones) and the evaluation is performed on various principally different test sets such as randomly generated data, network flow records, passwords and Facebook and Twitter data. The proposed approach is based on *linear genetic programming* (LGP) combined with a multi-objective NSGA-II algorithm, where the objectives are the number of collisions (after embedding the hash function to a hash table) and the execution time. As measuring the real execution time on a particular machine is time consuming (during the evolution), the execution time is estimated according to the number and type of instructions used by a particular candidate hash function. In order to estimate this value for modern processors, a specialized procedure is developed which considers not only the complexity of instructions, but also their scheduling on SIMD architectures. Evolved hash functions are compared in terms of quality of hashing and execution time with 8 human-designed and 2 evolved general-purpose hash functions available in the literature.

The rest of the paper is organized as follows. Section 2 briefly introduces the principles of hash functions and previous work on evolving hash functions. The proposed multi-objective method is introduced in Sect. 3. Section 4 describes our results from the experiments performed in order to evaluate the proposed method and compare resulting hash functions with existing solutions. Conclusions are given in Sect. 5.

2 Related Work

In this section, the principles of hash functions are presented and evolutionary approaches developed to the design of hash functions are briefly surveyed.

2.1 Hash Functions

A *hash function* is a mathematical function h that maps an input binary string (of length k) to a binary string of fixed length (l), $h : 2^k \rightarrow 2^l$, where $k >> l$. The output value is called *hash value* or simply *hash* [5]. 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 $x \neq y$. One of desirable properties of hash functions

is that similar input vectors produce completely different outputs. This is called the avalanche effect.

The most important application of hash functions is the *hash table* [6]. Based on the key (the input to the hash function) a particular row (index) of the table is activated and data are read/stored from/to a memory slot with that index. In order to handle collisions (different data mapped to the same index), a separate chaining method, cuckoo hashing, coalesced hashing and other techniques have been developed. In the case of the separate chaining method, a list of records having the same hash is operated for each index of the table. A newly entered data record is then stored to the first empty item of the list connected to the particular index. If there is at most one occupied record at index i then the time complexity of lookup is $O(1)$; if n records exist then the complexity is $O(n)$ for the i-th index.

The quality of non-cryptographic hash functions is given in terms of the collision resistance (good hash functions generate a minimum number of collisions), avalanche effect, distribution of outputs, execution time and table load factor (for a given memory size). 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 [7].

2.2 Hash Function Design

Non-cryptographic hash functions are mostly used in hash tables [6]. Other important applications are Bloom filters [8], geometric hashing [9], coherency sensitive hashing [10,11] etc. A common approach to the automatic hash function design is to apply a general construction procedure such as the Merkle-Damgård construction. The literature provides us with various implementations of general-purpose human-created hash functions including DJBHash [12], DEKHash [5], FVN (Fowler-Noll-Vo) [13], One At Time, Lookup3 [14], MurmurHash2, MurmurHash3 [15] and CityHash [16].

Evolutionary approaches have been primarily focused on the non-cryptographic hash function design and evolved with genetic algorithms [17], tree GP [18], linear GP [1], grammar evolution [19] and Cartesian GP [20]. They can further be divided according to the purpose, i.e. either application-specific hash functions [1,21] or general-purpose hash functions [18,22]. The difference lies in the input data size and the evaluation approach. The fitness function is usually based on measuring the avalanche effect [23,24] or the number of collisions [1,22].

3 Multi-objective Linear GP in Hash Function Design

As target hash functions are optimized with respect to the execution time, it is natural to represent them at the level of machine instructions. Hence, linear genetic programming in which candidate programs are represented as sequences of instructions for a register machine [25–27] is employed to evolve hash functions. In order to ensure a multi-objective design, LGP is connected with NSGA-II as introduced in [1]. This section deals with proposed representation and evaluation of candidate hash functions.

3.1 Candidate Program Processing

General-purpose hash functions are typically constructed using instructions such as logical functions (e.g. XOR, AND, OR), addition, multiplication and rotation. These instructions then define the instruction set for LGP. The initial population is generated randomly using these instructions. As the size of the input is arbitrary in the case of general-purpose hashing, it is necessary to partition the input stream into several blocks and process them sequentially. Since the loop responsible for reading the input is always present, it makes no sense to evolve it. We will evolve just the body of the loop. Figure 1 shows that a candidate hash function is called in each iteration to read a new block and combine it with intermediate results obtained from processing the previous blocks. Particularly in this case, 32 bits are copied from the input stream to register r[1] in each iteration. The resulting hash is produced to register r[0].

```
unsigned int candidateProgram (*input){
  r[0] = input[0];

  FOR (i = 1; i < length(input); i++){
    r[1] = input[i];
    <Candidate program>
  }
  return r[0] ⊕ (r[0] >> 32);
}
```

Fig. 1. Framework for candidate program evaluation. In this case, a 32 bit data input is read in each iteration.

3.2 Quality of Hashing

Inspired in [1], the quality of hashing is measured in terms of 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 the weighted number of collisions:

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

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

where s is the number of memory slots. This function clearly penalizes candidate hash functions showing many collisions at one slot. The objective is to minimize $f(h)$.

Algorithm 1. Execution time estimation

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

```
1  c ← RotateCodeOutputRegisterLast(p);
2  used-instructions = 0;
3  previous-used-instructions = 0;
4  used-registers ← Insert(output-register);
5  while previous-used-instructions == used-instructions do
6      previous-used-instructions = used-instructions;
7      used-instructions = 0;
8      c_p ← c;
9      while ⟨ i ← getLastInstruction(c_p) ⟩ do
10         if DestinationRegister(i) ∈ used-registers then
11             used-registers ← Insert(source-registers(i));
12             Increment(used-instructions);
13         remove instruction i from c_p;
14 return RotateBack(used-instructions);
```

3.3 Execution Time Estimation

As hash functions are very frequently called in some applications, it is important to optimize them with respect to the execution time. In order to capture features of modern processors supporting the Single Instruction Multiple Data (SIMD) paradigm, a method performing the execution time estimate takes into account not only the number of instructions and their type, but also their eventual parallel processing (which in principle reduces the execution time). In LGP, not all instructions of a candidate program contribute to the result. There are two types of redundant instructions. Firstly, the genotype may contain instructions whose output is not consumed by any other instruction (the so-called structural redundancy). Secondly, there could be instructions used in the phenotype, but not contributing to the resulting value. For example, if the code contains r[5] = r[1] + r[0]; r[5] = r[2] + r[0], the first instruction can be removed. The algorithm developed to estimate the execution time removes unused instructions in the first step and, in the second step, it identifies those instructions that can be executed in parallel.

Because we evolve the body of a loop and the evolved code is executed multiple times, we cannot use the same approach as [1] (i.e. analyzing the algorithm from the last to the first instruction and removing unused instructions) to estimate the execution time. The reason is that unused instructions of one iteration can be important in the next iteration. Hence, Algorithm 1, removing the unused instructions, has more steps. Firstly, the instructions of the candidate program have to be rotated to a state in which the output register of the hash function is at the last position of the program. The program is analyzed in rounds, until all used instructions are not marked. Then unused instructions can

be removed. Finally, the resulting code has to be rotated back, because the next step performs instruction scheduling and the order of instructions is important (see Algorithm 1). Example is presented in Fig. 2.

We exploit the instruction level parallelism [28] enabling to process multiple data with a single instruction. Modern CPUs can typically process 256 bits at once which means that eight 64-bit operations can be executed in one instruction instead of executing 4 instructions sequentially. As introduced in [1], instruction

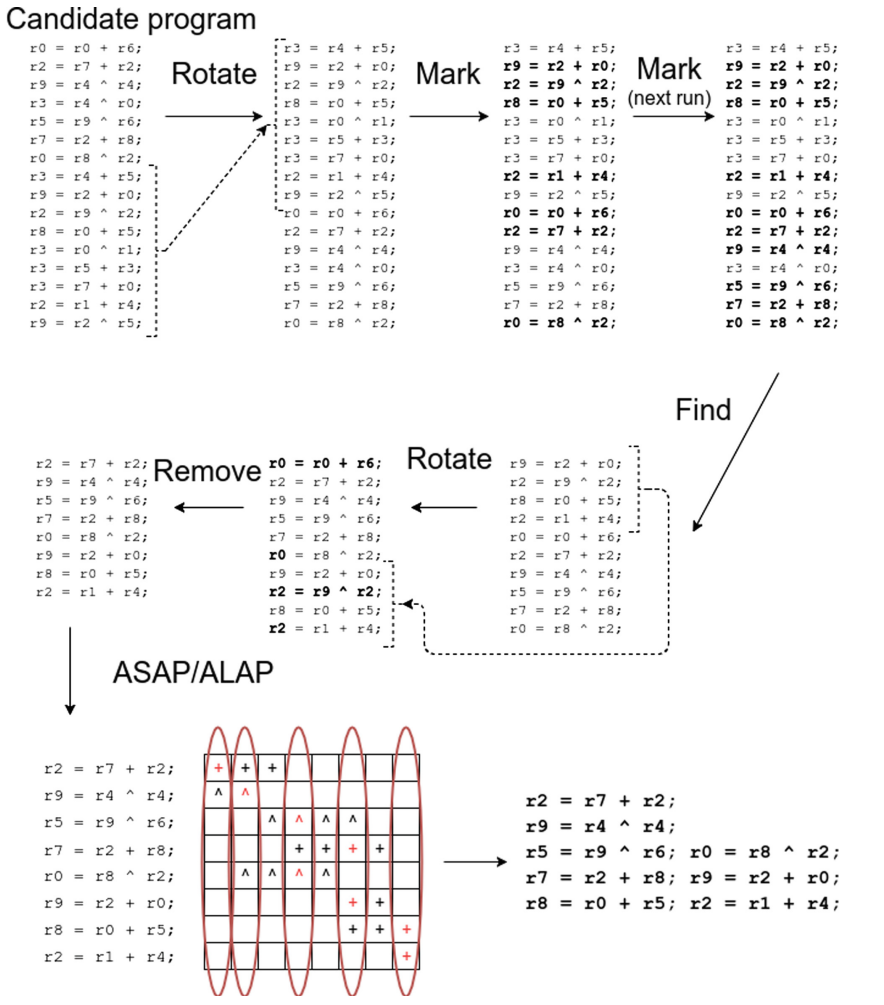

Fig. 2. Removal of unused instructions consists of rotating the candidate program to a configuration in which output register r0 is at position of the last instruction, identifying used instructions (in bold), removing unused instructions and rotating the code back. The optimized code is then scheduled for parallel execution. The final program consists of 5 steps in which 1, 1, 2, 2 and 2 instructions are executed in parallel.

```
unsigned int EvoHash1 (*input){
  r[0] = input[0];

  FOR (i = 1; i < length(input); i++){
    r[1] = input[i];
    r[8] = r[3] ⊕ r[1];
    r[5] = 0xA54FF53A;
    r[2] = r[5] + r[8];
    r[4] = r[1] * r[6];
    r[0] = r[2] | r[2];
    r[3] = r[4] | r[2];
  }
  return r0 ⊕ (r0 >> 32);
}
```

```
unsigned int EvoHash2 (*input){
  r[0] = input[0];

  FOR (i = 1; i < length(input); i++){
    r[1] = input[i];
    r[4] = r[2] ⊕ r[5];
    r[2] = r[1] + r[4];
    r[0] = r[0] + r[4];
  }
  return r0 ⊕ (r0 >> 32);
}
```

Fig. 3. Evolved hash functions that were selected from Pareto front in Fig. 4.

scheduling lies in determining when the instructions can be executed based on analyzing dependences among them. The ASAP (As Soon As Possible) and ALAP (As Late As Possible) routines are employed for this purpose. Figure 2 shows that in our example, the optimized 8-instruction program is finally executed in 5 steps in which 1, 1, 2, 2 and 2 instructions are executed in parallel.

3.4 Search Algorithm

A common version of LGP (with tournament selection, single-point crossover and mutation) is combined with NSGA-II [29]. According to [1], the maximum

Table 1. LGP parameters.

Parameter	Value
Population size	100
Crossover probability	90 %
Mutation probability	15 %
Program length	12
Registers count/type	8/64 b – int
Constants	{0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a, 0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19, 0x428a2f98, 0x71374491}
Instruction set (weight)	{ADD (1), MUL (3), XOR (1), OR (1)}
Tournament size	4
Maximum number of generations	100
Crossover type	One-point

program size is limited to 12 instructions. The function set contains those operations that are typical for the hash function design (XOR, AND, OR, addition, multiplication and right rotation). As multiplication is more complex than the remaining instructions, its execution time is counted with weight 3 in the programs. Common hash functions contain various "magic" constants. We extracted those appearing in the initial phase of hash function SHA-2 [30] and included them to the set of constants available in LGP. The setup for LGP is summarized in Table 1. NSGA-II is employed to find the best trade-offs between the number of collisions (according to Eq. 2) and estimated execution time for a training set (see Sect. 4).

4 Experiments and Results

This section describes the data sets used for evaluation, experiments and their analysis in terms of quality of hashing and execution time. Results will be compared with hash functions from the literature.

4.1 Data Sets

In order to evaluate candidate hash functions on different types of problems, we used (i) randomly generated data and (ii) real-world data coming from network flows, user passwords, and Facebook and Twitter posts.

We randomly generated the training data set (using a random text generator) in such a way that it contains 200,000 vectors with a random size ranging from 16 to 1024 characters. The best-evolved hash functions and the hash functions taken from the literature were then compared using 9 different randomly generated test data sets (Dataset1–9) whose parameters are summarized in Table 2.

In the case of real-world data, data sets Netset1–3 are formed from identifiers of network flows (source and destination IP addresses, source and destination ports and transport protocol). The size of each input vector is 96 bits (see details in [1]). The Passwords data set contains 10 million user passwords. Every passwords consists of 5 to 16 characters. Finally, Facebook and Twitter data sets contain 1 million posts from selected social network groups. These posts are in English, German, Hungarian, Czech and Slovak languages.

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 and evolved hash functions available in the literature (GPHash [23,24] and EFHash [22]). A 32-bit hash table is used for testing all functions. A direct comparison with [1] is possible only for the specific data sets used in [1]. Application-specific hash functions (XORhash, NSGAHash1, NSGAHash2, NSGAHash3, NSGAHash4, NSGAHash5, NSGAHash6, NSGAHash7 [1]) operate with a 96-bit input and produce a 16 bit hash value. Evolved hash functions produce a 32 bit hash value. The XOR folding is used for reduction from 32 to 16 bits.

Table 2. Data sets.

Name	Number of vectors	Length [bytes]
Dataset1	100,000	64
Dataset2	100,000	128
Dataset3	100,000	256
Dataset4	100,000	512
Dataset5	100,000	1024
Dataset6	100,000	2048
Dataset7	1,000,000	16 – 4096
Dataset8	1,000,000	16 – 4096
Netset1	20,000	12
Netset2	50,000	12
Netset3	100,000	12
Passwords	10,000,000	5 – 16
Facebook	1,000,000	3 – 280
Twitter	1,000,000	3 – 5000

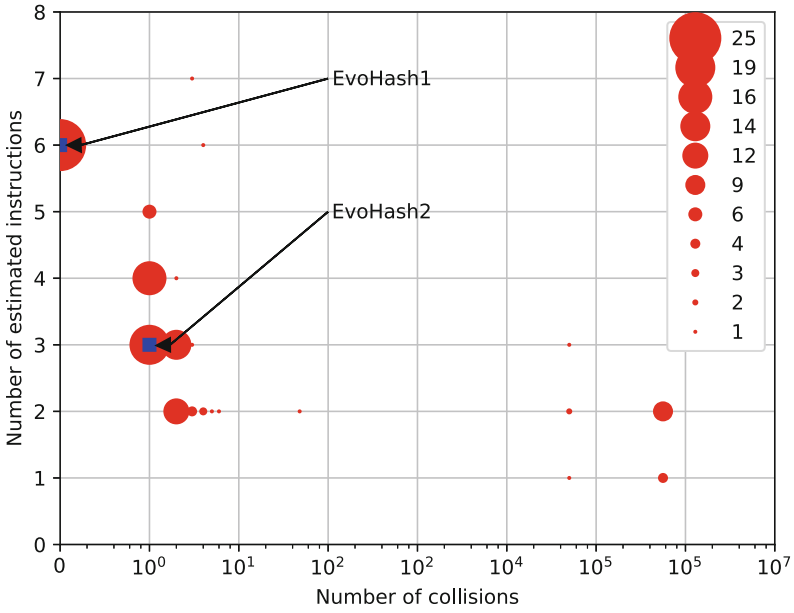

Fig. 4. Pareto fronts obtained from 100 independent runs of LGP. The size of the circle represents the number of identical solutions with the same properties. Selected hash functions (blue squares) are given in Fig. 3. (Color figure online)

4.3 Resulting Pareto Fronts

As we used the same parameters of LGP as [1], we do not report the impact of LGP parameters on the equality of evolution. The main focus is on a comparison of key parameters of evolved hash functions with existing hash functions.

We performed 100 independent runs of our multi-objective LGP and plotted in Fig. 4 parameters of all solutions appearing on the (100) final Pareto fronts. As many identical trade-offs were discovered in several (independent) runs, we plotted them using a circle whose diameter depends on the number of such cases. From all these designs, we selected two the most frequently occurring candidates (blue squares) and analyzed their properties in greater detail. EvoHash1 (see the C code in Fig. 3) produces zero collisions on the training data set, but includes relative many instructions. EvoHash2 (see the C code in Fig. 3) shows the best trade-off between the number of instructions and the number of collisions.

Since there are no clear outliers on Pareto fronts and the designs showing desired trade-offs are represented by larger circles (i.e. there are many good solutions), we can conclude that the proposed algorithm produces stable solutions. It can be seen in Fig. 4 that there are almost no solutions showing $10^1 - 10^4$ collisions. Our explanation for this behavior is that there are only a few discrete points for the second objective (the number of instructions) and these points are already covered by good solutions.

4.4 The Number of Collisions

The hash functions from the literature introduced in Sect. 4.2 were implemented in C programming language and compiled with the same compiler setting as evolved hash functions. All tests were then carried out with these implementations to ensure fair comparisons. The evaluation of all these hash functions was performed on an Intel Xeon E5-2620v3 processor running at 2.4 GHz.

Table 3. The number of collisions for randomly generated data sets.

Hash function	The number of collisions							
	DataSet1	DataSet2	DataSet3	DataSet4	DataSet5	DataSet6	DataSet7	DataSet8
DJBHash	**0**	3	**0**	***1***	***1***	3	132	116
DEKHash	60004	90000	90000	90000	90000	90000	122	118
FVNHash	**0**	4	***1***	***1***	***1***	**0**	115	122
One At Time	***1***	2	2	2	***1***	***1***	**108**	115
lookup3	***1***	**0**	**0**	2	***1***	2	122	111
Murmur2	***1***	***1***	***1***	**0**	3	3	125	126
Murmur3	2	**0**	2	***1***	***1***	3	***114***	111
CityHash	3	***1***	***1***	***1***	***1***	**0**	125	111
GPHash	***1***	***1***	***1***	***1***	**0**	**0**	115	**102**
EFHash	38137	53488	63353	64983	65119	65209	799933	799825
EvoHash1	2	2	2	***1***	***1***	***1***	133	116
EvoHash2	***1***	***1***	**0**	3	3	***1***	119	***108***

Table 3 gives the number of collisions for all randomly generated datasets for a 32 bit hash table. The best values are typed in **bold**; the second best values in ***bold-italic***. It can be seen that hash functions evolved by LGP produce a very similar number of collisions as other hash functions from the literature; except DEKHash and EFHash where many collisions are visible. From the point of view of the number of collisions, evolved hash functions are as good as the other hash functions. The same phenomenon can be observed for real-world data sets (see Tables 4 and 5).

4.5 The Execution Time and Performance

Tables 6, 7, 8 show the average execution time obtained from 50 independent runs of all hash functions on all data sets. The task is to compute a hash value for each vector of a given dataset. The evolved hash functions exhibit the shortest execution time in almost all cases. Similar parameters show Google's CityHash.

Table 4. The number of collisions for network data from [1].

Hash function	The number of collisions		
	NetSet1	NetSet2	NetSet3
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
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
EvoHash1	2849	15185	48652
EvoHash2	2821	14982	48695

Table 5. The number of collisions for real-world data sets.

Hash function	The number of collisions		
	Passwords	Facebook	Twitter
DJBHash	11663	247	137
DEKHash	14114	357	153
FVNHash	11845	115	115
One At Time	11590	105	138
lookup3	11567	119	107
Murmur2	11637	112	123
Murmur3	11589	103	***89***
CityHash	***11530***	122	122
GPHash	11634	117	113
EFHash	9983806	873270	824153
EvoHash1	11871	***23***	98
EvoHash2	**11469**	**10**	**1**

Evolved EvoHash2 is slightly faster (4%) than CityHash, but significantly faster (2x) than very popular Murmur hash 3.

Table 7 shows that the application-specific hash functions have a shorter execution time for the network data sets. But evolved hash functions are faster than the best conventional hash functions (CityHash, lookup3).

Finally, we compared all hash functions in terms of throughput that can be obtained by SMHasher [31]. This is a test suite designed to test performance properties of non-cryptographic hash functions. In the Bulk speed test (with

Table 6. The average execution time for randomly generated data sets.

Hash function	Execution time [ms]							
	DataSet1	DataSet2	DataSet3	DataSet4	DataSet5	DataSet6	DataSet7	DataSet8
DJBHash	19.56	32.914	45.311	72.31	126.081	231.675	2556.226	2554.123
DEKHash	12.907	19.352	28.141	46.975	81.419	156.839	1875.878	1872.019
FVNHash	17.354	31.694	48.371	83.761	155.702	294.259	3223.727	3220.844
One At Time	20.208	36.895	57.667	100.993	189.24	360.009	3918.302	3916.603
lookup3	12.867	22.685	28.403	42.581	72.585	125.851	1437.492	1433.961
Murmur2	12.06	20.332	25.718	36.065	60.202	102.426	1195.029	1190.402
Murmur3	12.863	21.622	27.796	40.367	68.557	119.167	1368.135	1363.745
CityHash	10.906	18.591	20.344	24.807	36.806	***54.535***	***683.363***	***679.325***
GPHash	25.497	47.418	80.294	147.286	283.533	550.774	5949.786	5948.746
EFHash	24.394	41.66	69.332	127.822	246.387	479.26	5237.982	5237.599
EvoHash1	***10.383***	***17.084***	***19.056***	***23.897***	***35.508***	55.838	685.604	681.327
EvoHash2	**10.385**	**17.411**	**19.022**	**23.825**	**53.132**	**37.334**	**659.185**	**656.647**

Table 7. The average execution time for network data from [1].

Hash function	Time [ms]		
	NetSet1	NetSet2	NetSet3
DJBHash	1.861	5.134	12.724
DEKHash	1.221	4.373	10.407
FVNHash	1.301	4.721	9.633
One At Time	1.769	5.290	12.352
lookup3	0.925	2.891	7.435
Murmur2	1.034	3.095	7.925
Murmur3	1.193	3.215	8.727
CityHash	0.960	2.625	7.407
XORHash	0.838	2.318	6.652
GPHash	1.865	4.671	12.558
EFHash	2.472	13.527	49.495
NSGAHash1	0.529	2.804	8.507
NSGAHash2	***0.527***	**2.072**	6.564
NSGAHash3	**0.514**	2.779	8.492
NSGAHash4	0.530	***2.073***	***6.219***
NSGAHash5	0.534	2.081	6.288
NSGAHash6	***0.527***	2.083	6.249
NSGAHash7	0.547	2.175	6.449
EvoHash1	0.802	2.569	7.455
EvoHash2	0.830	2.825	7.835

262144 byte keys), evolved hash functions EvoHash1 and EvoHash2 outperformed the remaining hash functions (Table 9).

Table 8. The average execution time for real-world data sets.

Hash function	Time [ms]		
	Passwords	Facebook	Twitter
DJBHash	5438.594	17.331	16.726
DEKHash	5067.882	13.240	13.119
FVNHash	5499.328	14.174	12.767
One At Time	6072.904	15.410	13.955
lookup3	4543.399	12.009	10.919
Murmur2	4464.339	11.723	10.774
Murmur3	4573.453	11.955	10.966
CityHash	4385.625	11.149	10.355
GPHash	6389.323	17.966	16.167
EFHash	5101.523	14.304	13.746
EvoHash1	**4268.402**	***10.895***	***9.996***
EvoHash2	***4277.341***	**10.832**	**9.954**

Table 9. Speed test according to SMHasher [31].

Bulk speed test – 262144-byte keys – MiB/sec								
Hash function	Alignment							
	0	1	2	3	4	5	6	7
DJBHash	1268.27	1271.40	1271.40	1271.40	1271.40	1271.40	1271.40	1271.40
DEKHash	1906.95	1907.01	1907.02	1907.01	1907.00	1907.06	1907.06	1907.05
FVNHash	953.63	953.63	953.63	953.63	953.63	953.63	953.63	953.63
OneAtTime	634.20	634.12	634.12	634.15	634.14	634.12	634.15	634.14
lookup3	2750.08	2735.18	2735.27	2735.29	2749.80	2735.26	2735.20	2735.14
Murmur2	3813.36	3780.15	3780.15	3780.15	3813.46	3780.25	3780.25	3780.25
Murmur3	7476.99	7332.31	7335.21	7332.47	7333.44	7334.75	7332.51	7334.79
CityHash	***15450.42***	14386.41	14370.53	14389.85	14390.17	14372.77	14385.49	14400.47
GPHash	475.67	475.68	475.68	475.69	475.69	475.68	475.68	475.69
EFHash	543.60	543.59	543.59	543.58	543.60	543.58	543.59	543.59
EvoHash1	15121.84	***14661.90***	***14662.12***	***14663.13***	***14662.58***	***14662.96***	***14662.41***	***14662.68***
EvoHash2	**17578.29**	**16726.21**	**16726.44**	**16725.27**	**16730.33**	**16726.50**	**16727.08**	**16728.04**

5 Conclusions

In this paper, we proposed and evaluated a multi-objective evolutionary design approach in which LGP is combined with NSGA-II algorithm in order to obtain general-purpose, ultra-fast and high-quality hash functions. This proposal addressed current needs of IT industry which seeks for high quality, but ultra fast hash functions. The fitness function was based on (i) the number of collisions with penalization for candidate hash functions producing many collisions and (ii) the execution time.

The best evolved hash functions were compared with 10 hash functions from literature. In terms of quality, evolved hash functions produce almost the same number of collisions as other good hash functions. In terms of the execution time and performance, a hash function improving parameters of a high quality conventional solution (CityHash) was discovered.

Our future work will be devoted to improving the design framework (in terms of supporting other objectives and accelerating the design process) and detailed testing of evolved hash functions in other real-world applications.

Acknowledgments. This work was supported by the Czech science foundation project 16-08565S. The authors would like to thank Dr. Martin Zadnik for his valuable comments to this research.

References

1. Grochol, D., Sekanina, L.: Multiobjective evolution of hash functions for high speed networks. In: Proceedings of the 2017 IEEE Congress on Evolutionary Computation, pp. 1533–1540. IEEE Computer Society (2017)

2. Dobai, R., Korenek, J., Sekanina, L.: Adaptive development of hash functions in FPGA-based network routers. In: 2016 IEEE Symposium Series on Computational Intelligence, pp. 1–8. IEEE Computational Intelligence Society (2016)
3. Kidoň, M., Dobai, R.: Evolutionary design of hash functions for ip address hashing using genetic programming. In: 2017 IEEE Congress on Evolutionary Computation (CEC), pp. 1720–1727. IEEE (2017)
4. Kocsis, Z.A., Neumann, G., Swan, J., Epitropakis, M.G., Brownlee, A.E.I., Haraldsson, S.O., Bowles, E.: Repairing and optimizing hadoop *hashCode* implementations. In: Le Goues, C., Yoo, S. (eds.) SSBSE 2014. LNCS, vol. 8636, pp. 259–264. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09940-8_22
5. Knuth, D.E.: The Art of Computer Programming, vol. 3 (1973)
6. Maurer, W.D., Lewis, T.G.: Hash table methods. ACM Comput. Surv. (CSUR) **7**(1), 5–19 (1975)
7. Pagh, R., Rodler, F.F.: Cuckoo hashing. In: auf der Heide, F.M. (ed.) ESA 2001. LNCS, vol. 2161, pp. 121–133. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44676-1_10
8. Song, H., Dharmapurikar, S., Turner, J., Lockwood, J.: Fast hash table lookup using extended bloom filter: an aid to network processing. SIGCOMM Comput. Commun. Rev. **35**(4), 181–192 (2005), https://doi.org/10.1145/1090191.1080114
9. Lamdan, Y., Wolfson, H.J.: Geometric hashing: a general and efficient model-based recognition scheme (1988)
10. Korman, S., Avidan, S.: Coherency sensitive hashing. IEEE Trans. Pattern Anal. Mach. Intell. **38**(6), 1099–1112 (2016)
11. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S.: Locality-sensitive hashing scheme based on p-stable distributions. In: Proceedings of the Twentieth Annual Symposium on Computational Geometry, SCG 2004, pp. 253–262. ACM, New York (2004), https://doi.org/10.1145/997817.997857
12. Bernstein, D.J.: Mathematics and computer science, https://cr.yp.to/djb.html. Accessed 31 Jan 2016
13. Fowler, G., Vo, P., Noll, L.C.: FVN Hash, http://www.isthe.com/chongo/tech/comp/fnv/. Accessed 31 Jan 2016
14. Jenkins, B.: A hash function for hash table lookup, http://www.burtleburtle.net/bob/hash/doobs.html. Accessed 31 Jan 2016
15. Appleby, A.: Murmur hash functions, https://github.com/aappleby/smhasher. Accessed 31 Jan 2016
16. Pike, G., Alakuijala, J.: Introducing cityhash (2011)
17. Safdari, M., Joshi, R.: Evolving universal hash functions using genetic algorithms. In: International Conference on Future Computer and Communication, ICFCC 2009, pp. 84–87, April 2009
18. Estebanez, C., Saez, Y., Recio, G., Isasi, P.: Automatic design of noncryptographic hash functions using genetic programming. Comput. Intell. **30**(4), 798–831 (2014)
19. Berarducci, P., Jordan, D., Martin, D., Seitzer, J.: Gevosh: using grammatical evolution to generate hashing functions. In: MAICS, pp. 31–39 (2004)
20. Widiger, H., Salomon, R., Timmermann, D.: Packet classification with evolvable hardware hash functions – an intrinsic approach. In: Ijspeert, A.J., Masuzawa, T., Kusumoto, S. (eds.) BioADIT 2006. LNCS, vol. 3853, pp. 64–79. Springer, Heidelberg (2006). https://doi.org/10.1007/11613022_8
21. Kaufmann, P., Plessl, C., Platzner, M.: EvoCaches: application-specific adaptation of cache mappings. In: Adaptive Hardware and Systems (AHS), pp. 11–18. IEEE CS (2009)

22. Karasek, J., Burget, R., Morskỳ, O.: Towards an automatic design of non-cryptographic hash function. In: 2011 34th International Conference on Telecommunications and Signal Processing (TSP), pp. 19–23. IEEE (2011)
23. Estébanez, C., Hernández-Castro, J.C., Ribagorda, A., Isasi, P.: Finding state-of-the-art non-cryptographic hashes with genetic programming. In: Runarsson, T.P., Beyer, H.-G., Burke, E., Merelo-Guervós, J.J., Whitley, L.D., Yao, X. (eds.) PPSN 2006. LNCS, vol. 4193, pp. 818–827. Springer, Heidelberg (2006). https://doi.org/10.1007/11844297_83
24. Estebanez, C., Hernandez-Castro, J.C., Ribagorda, A., Isasi, P.: Evolving hash functions by means of genetic programming. In: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, pp. 1861–1862. ACM (2006)
25. Brameier, M., Banzhaf, W.: Linear Genetic Programming. Springer, New York (2007). https://doi.org/10.1007/978-0-387-31030-5
26. Oltean, M., Grosan, C.: A comparison of several linear genetic programming techniques. Complex Syst. **14**(4), 285–314 (2003)
27. Wilson, G., Banzhaf, W.: A comparison of cartesian genetic programming and linear genetic programming. In: O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcázar, A.I., De Falco, I., Della Cioppa, A., Tarantino, E. (eds.) EuroGP 2008. LNCS, vol. 4971, pp. 182–193. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78671-9_16
28. Wall, D.W.: Limits of Instruction-level Parallelism, vol. 19. ACM, New York (1991)
29. Deb, K., Agrawal, S., Pratap, A., Meyarivan, T.: A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In: Schoenauer, M., Deb, K., Rudolph, G., Yao, X., Lutton, E., Merelo, J.J., Schwefel, H.-P. (eds.) PPSN 2000. LNCS, vol. 1917, pp. 849–858. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45356-3_83
30. NIST: Secure hashing, https://csrc.nist.gov/projects/hash-functions. Accessed 10 Oct 2017
31. Appleby, A.: Smhasher, https://github.com/aappleby/smhasher. Accessed 1 Nov 2017