BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2016
http://excel.fit.vutbr.cz

Evolutionary design of domain specific non-cryptographic hash functions

Marek Kidoň*

Abstract
Hash functions are inseparable part of modern computer world. Fast associative arrays so popular among computer programmers for their robustness and simplicity, are based on them. Their performance greatly depends on their design and although their roots are deep in the past, the topic of designing a well performing hash function is still often discussed today. There is currently a plenty of exceptional implementations of generic hash functions and their numbers are rising. Such functions are not constrained to a concrete set of inputs, they perform on any input. On the other hand, there are use cases when input domain is known in advance. In such case, there is a room for an improvement by designing specific hash function thus reaching better level of performance in comparison with a generic hash function. However designing a hash function is not a trivial task. There are no rules, standards or guides. In case of manual design the hash function author has to rely on his/her knowledge, experience, inventiveness and intuition. This fact opens up a space for different techniques such as evolutionary algorithms, an unconventional approach to solve certain problem inspired by the process of species reproduction. In this paper hash functions are designed for the domain of IP addresses using genetic programming. Genetic programming algorithm parameters are accurately chosen so the evolved functions will perform on its best. The experiments proved that the developed hash functions are better in comparison with generic hash functions terms of both speed and performance.

Keywords: Hash Function — Genetic Programming — Evolution Design

Supplementary Material: *N/A*

*xkidon00@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Hashing is used in fast associative array implementations among other applications. Its concept seems to be ageless since there has been a lot of recent progress. For example popular hash function *MurmurHash2* [1] found its application at Apache Software Foundation and has already been replaced with *MurmurHash3* [2]. In 2010 Google developed the *CityHash* [3] function family which mainly focuses on hashing strings. It has been strongly influenced by *MurmurHash2* but its main concern is speed and portability. Yet in 2014 Google released a new family called the *FarmHash* [4] focusing on hashing strings as well as other data structures.

All previously stated hashes are generic and thus have been designed to work for any input. But sometimes the input domain is known in advance. In such case we can design custom hash function for the input domain and reaching better performance levels. Considering associative arrays the hash function is the critical point especially in performance demanding applications. In our work, we use 4 subsets of the Internet Protocol (IP) address range, containing 8192 addresses each. Our task is to automatically design well performing hash function for each of these subsets. The goal of this work is to design a single hash by an evolutionary algorithm for each subset that outperforms conventional generic hash functions in terms of collision resistance and speed.

A good solution is such solution which outperforms solution designed by a human experts in hashing in both speed and collision resistance criteria. Of course there is the concept of perfect hashing [5], but our work aims at use cases such as field programmable gate array, where perfect hashing cannot be used.

Evolution design has changed the way people think about problems and their solutions [6]. A wide variety of problems can be viewed as a state space search problems where evolutionary algorithms do excel. Designing a hash function is no exception [7, 8]. Each hash represents a single point (feasible solution) in a huge state space. We will design hash functions using genetic programming. Both genetic programming individuals and hash functions can be represented by an expression. This is the core fact that we will exploit in our pursuit of designing well performing hash functions.

Evolution design of domain specific hash function is a rare topic. It is suitable for cases where perfect hashing cannot be used and conventional generic hash functions are too slow and suffer from high collision rate. In such case evolutionary designed domain specific hashes could outperform generic ones by a significant margin in terms of collision resistance and by great margin in terms of speed.

2. Existing Method Approaches

Evolution design of non-cryptographic hash function has been a interesting topic in the last decade. Majority of previous work focuses on development of generic hashes [7, 8]. Both evolutionary optimization and evolutionary design have been considered. A similar work to ours is [7]. It is similar in terms of goals and used technique but uses evolutionary optimization instead. The state space has been dramatically constrained in the beginning since the author optimizes parameters a,b of the following hash function $h_{a,b}(k)$:

$$h_{a,b}(k) = ((ak+b) \ mod \ p) \ mod \ N \tag{1}$$

where p is a prime number and $N-1$ is the maximum output slot. "The results are promising but the methodology is questionable" [8]. Since the input set is chosen randomly, why not use elements of the input directly? Also the collision resistance criteria is not well suited for evolving generic hash functions since it is very input dependent.

Probably the closest work to our problem is the [8] where the authors also selected genetic programming as evolutionary platform and even the parameter choice is similar. Choosing avalanche effect properties as basis for measuring the fitness of evolved individuals which seems like a good choice. The avalanche effect is a desirable hash function property reflecting the rate of change of output on change of the input. Even a small change of input should produce great change on output. Avalanche effect is input independent which is a very positive feature in this case. They tested their system on various key-sets and reached promising results. They state that their system is very robust since a change of evolution parameters has low or no impact on the quality of solution but such behavior can result in degradation of the algorithm to random search. Also they state they are automatically designing hash functions. This does not seem to be entirely true because they are merely optimizing mixing components of the Merkle–Damgård scheme [9], a generally used construction for (primarily) cryptographic hash function.

3. Design and Implementation

In this section we will explain high level design of evolved hashes and decisions we made in choosing genetic programming algorithm parameters. Genetic programming belongs among evolutionary algorithms where candidate solutions are represented by computational trees. These trees will directly represent our hash functions. A group of candidate solutions form a generation. The algorithm progresses throughout its run by repetitively breeding new generations using genetic operators. The run ends if a good enough solution was found or maximum generation was reached.

As stated in [10], we need to make several *preparatory steps* and wisely choose the following:

- *terminal set*, the set of functions of zero arity,
- *non-terminal set*, the set of function of arity greater than zero,
- the *fitness* measure,
- termination criterion,
- set of auxiliary parameters controlling the run.

Unusually the simplest of these is to define the fitness function. Since we are developing domain specific functions, we will stick with data dependent fitness measure. We will use collision resistance and we will be minimizing amount of collisions on given data set. The fitness function is defined as follows:

$$f_{set,h} = collisions/size \tag{2}$$

a ratio of size of the input set and number of collisions hash function produced on given input set. An important criteria of hash function is their speed. As we will see later, all of our hash functions are inherently fast and very similar to each other in terms of speed. Therefore we are not measuring this.

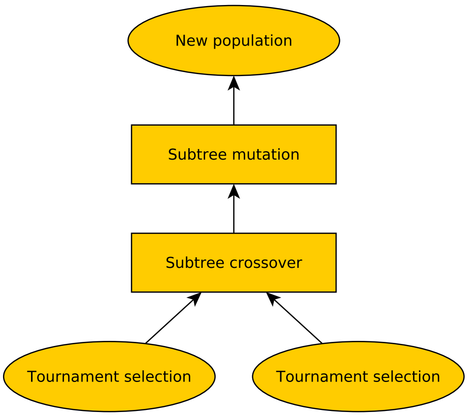

Figure 1. Basic reproduction pipeline.

Common operations found in hash functions such as *MururHash* or *lookup3* are the multiplication ($*$), addition ($+$), left or right rotations ($\lll, \ggg$), left or right shifts ($\ll, \gg$) , XOR ($\wedge$) and ones' complement ($\neg$). Although these are the most common sometimes we can encounter more exotic functions such as minus, modulus, bitwise OR and bitwise AND. We choose our non-terminal set to be as follows:

$$NTS = \{*, +, \wedge, \ggg\} \tag{3}$$

The third step but not less important is to select the terminal set. We work with IP version four addresses. Each address consists of four octets, eight bit each. We add each octet to our terminal set and denote them o_0, o_1, o_2, o_3. State of the art hash functions usually use some kind of magic constant to introduce an additional randomness. In our case we use ephemeral random constants. Such constants are randomly generated upon creation and fulfill basic genetic programming requirements such as, they can be mutated or regenerated. In our case, we will use unary function (denoted $\Re$) which selects arbitrary prime number from list of all prime numbers in the $\langle 2, 2N \rangle$ interval where N is size of the input dataset. Our resulting terminal set is:

$$TS = \{o_0..o_3, \Re\} \tag{4}$$

The algorithm is allowed to terminate if the limit of 100 000 candidate solution evaluations has been reached or a perfect solution has been found. Perfect solution is solution with fitness equal to zero or in other words our candidate solution did not produce any collisions at all. This is however unlikely to happen.

Finally in the Figure 1 we can see our reproduction pipeline which together with Table 1 forms all remaining control parameters.

Table 1. Summary of genetic programming algorithm setup.

Parameter	Value
Evaluations	100000
Population size	512
Maximum depth	6
Non-terminal set	$\{*, +, \wedge, \ggg\}$
Terminal set	$\{o_0..o_3, \Re\}$
Initialization method	Ramped half-and-half
Initial maximum depth	6
Initial minimum depth	2
Selection	Tournament
Tournament size	7
Crossover	Subtree
Root selection rate	0.0
Leaf selection rate	0.1
Node selection rate	0.9
Mutation	Subtree
Rate	0.1
Elitism	Yes
Rate	0.05

4. Experiments and Results

Each experiment in this section is based on an average of 50 independent runs. We will compare our results with *Murmurhash3*, *CityHash* and *FarmHash*. These functions (families) are state of the art hash functions used in the modern computer world or their latest versions/modifications. Great example is the *MurMurHash3* which is currently used in various open-source projects such as the *libstdc++*, *nginx* or Apache *Hadoop*.

We have chosen *MurmurHash3* as a reference hash to compare it with the results of the first experiment. In the Figure 2 we can see that evolved functions are better performing than the human-created *MurmurHash3*. An interesting experimental result is that even in generation zero there exists at least one solution that performs better than all of these conventional hashes. Genetic programming run rapidly converges in early generations and stagnates in later generations [10]. We can see that this is our case exactly. In theory collision resistance performance could be enhanced by incorporating a collision resolution such as Cuckoo hashing mechanism [11] where two or more hash functions are used. Achieved results are better than one of the latest results [12] based on the evolutionary algorithm and Cuckoo hashing approach. Below we can se a sample hash function evolved using the proposed IPHash system.

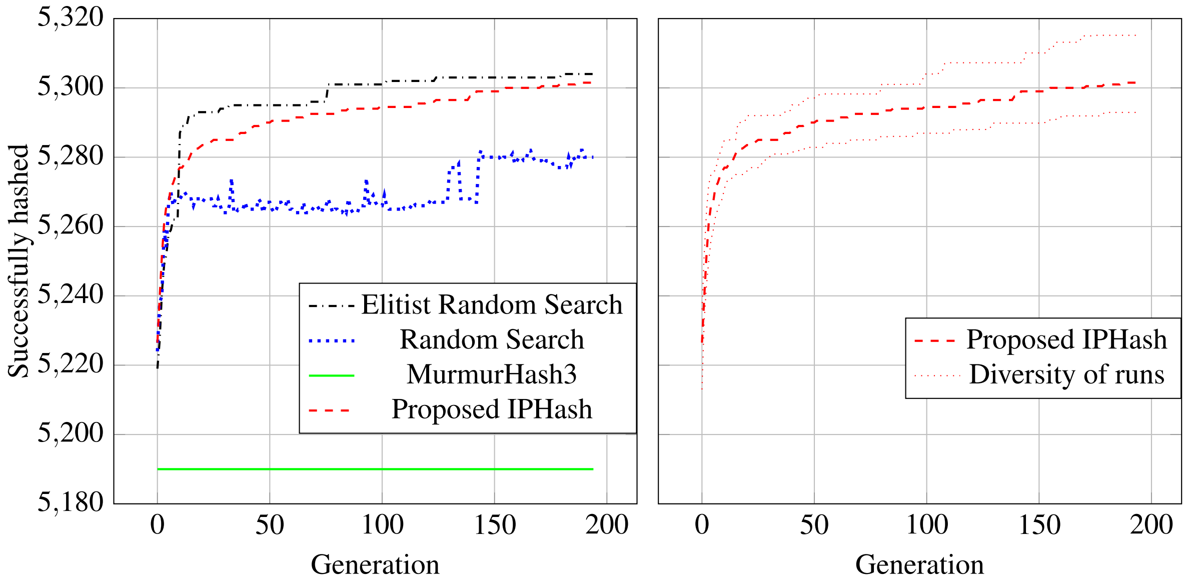

Figure 2. Evolved hash functions in comparison with conventional hash functions in terms of collision resistance.

$$f_{o_0,o_1,o_2,o_3} = (((\Re \wedge (o_1 \wedge o_3)) + (o_2 * \Re)) + ((\Re \ggg o_1) \ggg (((o_1 * o_0) \ggg o_1) * (\Re \ggg o_1)))) \quad (5)$$

In the same figure two versions of random search algorithm are put in comparison. The elitist random search performs even better than our genetic programming algorithm, however we should keep in mind, that the random search algorithm has been greatly influenced by the computational tree, terminal and non-terminal sets we have chosen. This outcome provides us with interesting conclusion that the state space as described by selected encoding and collision ratio fitness is more suitable for random search rather than genetic programming.

In the second experiment have investigated the impact of various population sizes and individual depths in zero generation. Also we have studied how various terminal and non-terminal sets affect resulting fitness values.

In the first part, we have selected our population sizes from the set of powers of two. Smallest population size was set to eight individuals, biggest to 1024 individuals. Initial depth will be selected from interval $\langle 2,6 \rangle$ with step of size 1. All other parameters remain unchanged. As we can see in the Figure 3 with exception of very shallow individuals and small generations, results are stable. The best individual that managed to hash 5314 addresses settled at population size 64 and depth 3. However bigger populations or deeper trees offer similar collision numbers and the deviations seem insignificant. On the other hand we should keep in mind that in terms of high performance applications difference of ten collision might become very significant.

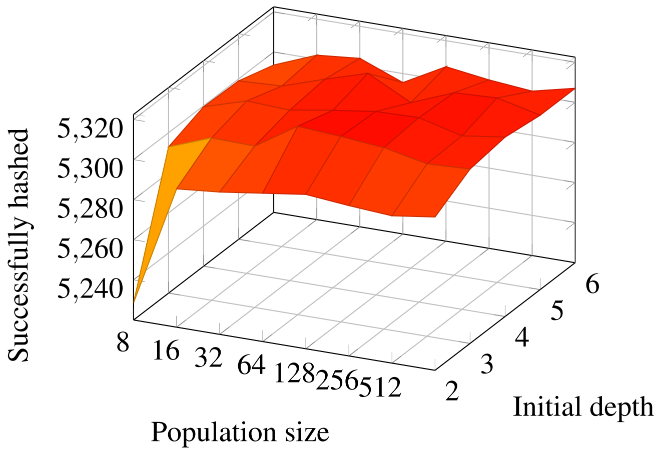

Figure 3. Number of successfully hashed addresses in relation to population size and initial depth.

In the second part we have chosen several alternatives to the basic setup (see Table 2) to be executed and their output studied. As expected the loss of additional randomness in form of ephemeral random constants negatively influenced the output. However adding ones' complement did not improve results in spite of the fact that the ones complement and XOR forms a complete logic set thus being able to compute any possible logic function. The addition is popular among hashing experts but as expected [8] its removal didn't have any significant influence on the results. The rotation (shift) operators is always part of the core of any good hash algorithm. The more interesting is the result of rotation removal experiment where resulting individual quality didn't suffer as significantly as expected. Finally the removal of both rotation and multiplication did have significant negative influence on average best fitness.

Table 2. Effect of various terminal and non-terminal sets on resulting fitness.

Terminal set	Function set	Average best
$\{o_0..o_3, \Re\}$	$\{*, +, \wedge, \ggg\}$	5307
$\{o_0..o_3\}$	$\{*, +, \wedge, \ggg\}$	5291
$\{o_0..o_3, \Re\}$	$\{*, +, \wedge, \neg, \ggg\}$	5300
$\{o_0..o_3, \Re\}$	$\{*, \wedge, \ggg\}$	5306
$\{o_0..o_3, \Re\}$	$\{+, \wedge, \ggg\}$	5290
$\{o_0..o_3, \Re\}$	$\{*, +, \wedge\}$	5298
$\{o_0..o_3, \Re\}$	$\{+, \wedge\}$	5065

In the final experiment we have compared evolved hashes with generic ones in terms of their computational size as well as summarized our results and enriched them with results on the remaining IP data sets.

Table 3 summarizes results reached so far in terms of hash function performance on all four IP data sets. We can also see the size of all used functions. The size expresses a number of elementary instructions in hash function. All hashes were compiled with *gcc* compiler version *4.8.3* on *x86_64* Linux system. Optimization flag level three was enabled. Results were obtained with *readelf* utility. The results may (and will differ) on different architecture with different compilers and setup. We can see that the proposed IPHash produces extremely compact functions for all IP data sets. We need to keep in mind that generic hash functions almost always contain a cycle so the number of executed instruction would get even higher.

5. Conclusions

This work focused on evolutionary design of hash functions for the domain of IP addresses. Hash functions are widely used basis for fast associative arrays. Their performance greatly depends on their design and although their roots are deep in the past, the topic of designing a well performing hash function is still often discussed today. The hash functions developed in or work outperformed *MurmurHash3*, *FarmHash* and *CityHash*. All of them are considered exceptional implementations of generic hash functions.

In terms of collision resistance we managed to outperform conventional hash functions by hashing at least a hundred keys more. In terms of speed evolved domain specific hashes are 3 to more than 10 times smaller then generic hash function.

We have shown that we can apply evolutionary design on unconventional problems and reach good results. We have researched evolutionary design applications towards a new direction and outperformed existing solutions in these fields.

Results of this work could lead to additional research in both genetic programming and hash functions. In the future work incorporating some sort of collision resolution mechanism such as open addressing or Cuckoo hashing might significantly improve the performance. Also the searched state space is wast. Using one of generally used hashing schemes might reduce the space towards better solutions.

Acknowledgements

I would like to thank my supervisor Roland Dobai for his help.

References

[1] Austin Appleby. Murmurhash2 webpage, 2011. https://sites.google.com/site/murmurhash/, [Online, accessed: 4. 5. 2016].

[2] Austin Appleby. Smhasher and murmurhash3 webpage, 2016. https://github.com/aappleby/smhasher/wiki/, [Online, accessed: 3. 5. 2016].

[3] Geoff Pike. Cityhash: Fast hash functions for strings. Standford university class slides, October 2012.

[4] Geoff Pike. Google open source blog, 2014. http://google-opensource.blogspot.cz/2014/03/introducing-farmhash.html, [Online, accessed: 4. 10. 2016].

[5] Bohdan S. Majewski Zbigniew J. Czech, George Havas. Fundamental study: Perfect hashing. pages 1–143, 1997.

[6] Lukáš Sekanina. *Evoluční hardware*. Academia, 2009.

[7] Mustafa Safdari. Evolving universal hash functions using genetic algorithms. In *Proceedings of the 11th Genetic and Evolutionary Computation Conference*, pages 2729–2732, New York, 2009.

[8] César Estébanez, Yago Saez, Gustavo Recio, and Pedro Isasi. Automatic design of noncryptographic hash functions using genetic programming. *Computational Intelligence*, 30(4):798–831, 2014.

[9] Ivan Damgård. A design principle for hash functions. In *CRYPTO '89: Proceedings of the*

Table 3. Effect of various terminal and non-terminal sets on resulting fitness.

	Set 1		Set 2		Set 3		Set 4	
Algorithm	Best	Size	Best	Size	Best	Size	Best	Size
MurMurHash3	5190	213	5190	213	5206	213	5206	213
CityHash	5180	926	5156	926	5171	926	5155	926
FarmHash	5227	788	5199	788	5203	788	5158	788
IPHash	5352	73	5352	58	5342	86	5298	79

9th Annual Inter- national Cryptology Conference on Advances in Cryptology, pages 416–427. Springer, 1989.

[10] Riccardo Poli, William B. Langdon, and Nicolas F. McPhee. *Genetic Programming An Introductory Tutorial and a Survey of Techniques and Applications*. Lulu Enterprises, 2008.

[11] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[12] Roland Dobai and Jan Kořenek. Evolution of non-cryptographic hash function pairs for fpga-based network applications. In *2015 IEEE Symposium Series on Computational Intelligence*, pages 1214–1219. Institute of Electrical and Electronics Engineers, 2015.