Fast Reconfigurable Hash Functions for Network Flow Hashing in FPGAs

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*—Efficient monitoring of high speed computer networks operating with a 100 Gigabit per second (Gbps) data throughput requires a suitable hardware acceleration of its key components. We present a platform capable of automated designing of hash functions suitable for network flow hashing. The platform employs a multi-objective linear genetic programming developed for the hash function design. We evolved high-quality hash functions and implemented them in a field programmable gate array (FPGA). Several evolved hash functions were combined together in order to form a new reconfigurable hash function. The proposed reconfigurable design significantly reduces the area on a chip while the maximum operation frequency remains very close to the fastest hash functions. Properties of evolved hash functions were compared with the state-of-the-art hash functions in terms of the quality of hashing, area and operation frequency in the FPGA.**

I. Introduction

Current high-speed computer networks can achieve a 100 Gigabit per second (Gbps) throughput and even 400 Gbps links will be available in the near future. At these speeds, detailed packet processing becomes a challenging problem. Fast packet processing is especially important in *network security* and *monitoring systems*, where any packet unseen by the monitoring system because of the system's insufficient performance can affect the quality of monitoring or disallow the detection of security threats. In order to achieve a 100 Gbps throughput, every packet has to be processed in less than 7 ns. It means that a single CPU core can only execute a few instructions to perform this job, which is far from needed. Hence, application-specific hardware accelerators have been developed to provide sufficient performance.

This paper deals with an automated design of ultra-fast *hash functions* that are crucial in these accelerators. In particular, hash functions will be developed for the *software defined monitoring* (SDM) platform. SDM performs network monitoring and analysis using relatively simple (and so fast) configurable circuits implemented in a *field programmable gate array* (FPGA). These circuits are configured by means of a *software application* whose purpose is to offload all time-critical packet processing tasks to hardware and perform only sophisticated analysis and other tasks that are not suitable for the hardware acceleration.

In SDM, the network traffic is analyzed at the level of *network flows*. A network flow is a sequence of packets from a source device to a destination, for example, a network flow can contain a specific transport connection or a media stream. One flow is defined by five parameters within a certain time period: source and destination IP address, source and destination port and transport protocol. These parameters will be referred to as a *flow identifier*. The role of hashing is to assign a memory slot (containing the data of a given flow) to the flow identifier extracted from network traffic.

The objective of this work is to develop and evaluate new hash functions suitable for network flow hashing in the FPGA. We will also explore possibilities of developing the reconfigurable hash functions whose implementation is motivated by recent attacks on traffic monitoring systems that use a hash function to distribute the network traffic (i.e. flow processing) on several cores. If the attacker can reveal how the network traffic is distributed, (s)he can generate a specific traffic from some IP addresses (and so flows) in such a way that (almost) all traffic is intentionally directed by the hash function to one core, the core becomes overloaded, some flows are dropped and thus remain invisible for security monitoring. However, if a reconfigurable hash function is supported, another configuration of the hash function can quickly be activated when one core becomes overloaded. This will change the unwanted workload distribution to the original status and keep the monitoring system working. In order to minimize the time spent in the less secure configuration, the system has to be adapted at the hardware level.

The proposed solution will be developed in the following steps. (i) We will introduce a genetic programming (GP) based system implemented for the evolutionary design of desired hash functions. (ii) Hash functions evolved with this system will be implemented in an FPGA, evaluated on several data sets and compared with conventional hash functions in terms of the quality of hashing, the area used in the FPGA and the maximal operation frequency. (iii) Finally, we will propose and evaluate a new reconfigurable hash function that combines selected parts of evolved hash functions in order to reduce the implementation cost.

The rest of the paper is organized as follows. Section II introduces the area of hash functions and their design, including the evolutionary hash function design. Section III presents a platform capable of automated evolutionary designing of hash functions suitable for network flow hashing. The approach utilized for the FPGA implementation of hash functions that were evolved by means of the platform is presented in Section IV.

978-1-5386-7753-7/18/$31.00 © 2018 IEEE

This section also deals with the development and experimental evaluation of a reconfigurable hash function. Conclusions are given in Section V.

II. 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* [1]. If $h(x) = h(y)$, where x and y are two inputs and $x \neq y$, the so-called *collision* is reported. Next section will describe one of the collision-handling methods that we employ in our application.

We will only deal with non-cryptographic hash functions in this paper. In the case of cryptographic hash functions, additional requirements (such as a pre-image resistance) are imposed on them, but these requirements are not relevant in our context.

A. Hash table

Fig. 1 shows how hash function h is used in a *hash table*, which is a data structure implementing an associative array [2]. Based on the input data (a key), the hash function computes a hash, i.e. an index into the array of slots, where the desired data can be found. Ideally, the hash function will address a unique slot, but collisions have to be handled in real-world applications. For this purpose, the separate chaining method, cuckoo hashing, coalesced hashing and other techniques have been developed. In the case of the *separate chaining method*, a linear list of records having the same hash is constructed and managed for each index of the table. If there is at most one occupied record at index i then the time complexity of lookup is constant; otherwise, it is linear with respect to the number of records at a given index.

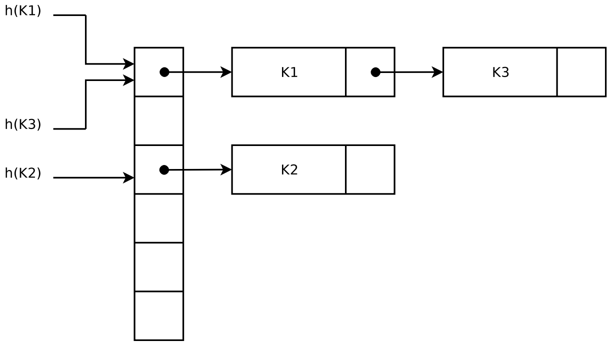


Fig. 1. Hash table with separate chaining.

The quality of non-cryptographic hash functions is evaluated based on their collision resistance (good hash functions should generate a minimum number of collisions), the avalanche effect (similar input vectors should produce completely different outputs), the distribution of outputs, the execution time and the table load factor (for a given memory size).

B. Design techniques

If a hash function is needed for a given application, the designer can either choose one of *general-purpose* hash functions available in the literature (such as DJBHash [3], DEKHash [1], FVN (Fowler-Noll-Vo) [4], One At Time, Lookup3 [5], MurmurHash2, MurmurHash3 [6] and CityHash [7]) or develop a new *application-specific* hash function.

Hash functions are usually designed by applying a general construction procedure such as the Merkle-Damgård construction [8]. However, a lot of approaches based on the evolutionary design principles have been introduced in recent years. Their main advantage is that they are capable of producing high-quality hash functions optimized for a given application domain. Hash functions were evolved with genetic algorithms [9], tree GP [10], linear GP [11], [12], grammar evolution [13] and Cartesian GP [14]. Both scenarios – application-specific hash functions (see, e.g., [15], [12], [16], [17]) and general-purpose hash functions (see, e.g., [10], [18]) – were addressed in the literature.

The fitness function is usually based on measuring the avalanche effect [19], [20] or the number of collisions [12], [18]. The execution time optimization has been explicitly addressed in [11], [12], where the hash function design was formulated as a multi-objective design problem.

C. Hashing in FPGAs

FPGA implementations of adaptive hash functions were developed for various network applications such as network routing [21], caching [22] and IP filtering [23]. For example, the hash function for IP filtering computes 12-bit hashes in 43 clock cycles for 32-bit inputs. Because of pipelined processing, one hash can be produced in each clock cycle, which gives 260 million hashes per second, i.e. 3.8 ns per hash [23]. This is more than sufficient for 400 Gbps links.

For comparative purposes of this paper, we implemented in FPGA two hash functions: XORHash and SipHash. The XORHash was developed for hashing of the network flows. It is based on the so-called *xor folding*, in which the components of the flow identifier are shifted by a predetermined number of bits and then summed by means of the xor function [24]. These implementations lead to high-speed pipelined structures. SipHash is a family of pseudorandom functions optimized for short inputs. Target applications include network traffic authentication and hash-table lookups [25]. A VHDL implementation is available at https://131002.net/siphash

III. Platform for hash function design

In our previous work, we developed a platform for evolutionary design of hash functions that are suitable for network flow hashing within the SDM concept [12]. The objective was not only to evolve high quality hash functions for this application, but also to optimize the execution time as the hash function is called very often.

A. Network flow hashing

The hash function is constructed for the hash table in which the collisions are handled with the separate chaining method. In IPv4, a network flow is defined using 104 bits representing the source IP address (32 b), the destination IP address (32 b), the source port (16 b), the destination port (16 b) and the transport protocol (8 b). In order to reduce the execution time, these inputs are processed in parallel, i.e. the hash function would consume 104 input bits. We proposed to reduce the dimension of the input vector to $3 \times 32 = 96$ 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 [11]

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

As the real traffic especially contains two types of transport protocol (TCP and UDP), there is not a significant loss of information using this reduction of the input vector. In addition, the input vector fits into three 32 bit registers which makes its processing straightforward on a common 32 bit processor. Finally, the resulting hash is represented on 16 bits.

B. Linear genetic programming

Linear genetic programming (LGP) [26], [27], [28] evolves computer programs that are represented as sequences of instructions for a register machine. The input and output program values are stored in the registers or in an external data memory. In our case, no external memory is needed because the 96 bit input can be stored in three registers (r0, r1 and r2) and the resulting hash is in the register r0. The remaining registers are initialized to 0. The number of registers available in the register machine is constant. Each instruction is typically represented by the instruction code, destination register and two source registers, for example, [xor, r4, r1, r2] is representing the operation r4 = r1 xor r2. Based on many experiments, a very restricted instruction set containing the addition, multiplication, logical XOR and right rotation was employed in our experiments. In some experiments, multiplication is even avoided to reduce the execution time of the resulting hash function. The impact of (not)supporting the multiplication on the execution time and quality of hashing was analyzed in [11]. The program size is restricted to contain only several instructions (usually less than 20) in order to force LGP to create short programs.

LGP typically operates with 200 individuals in the population, one-point crossover with probability 90 %, mutation probability 15 % and tournament selection [11], [12]. The register machine contains eight 32 bit registers. In our experiments, 1000 generations are produced in each LGP run.

C. Fitness functions

Two objectives can be optimized by the proposed platform: the quality of hashing and the execution time of a hash function.

In order to evaluate a candidate hash function, it is executed with a set of flow identifiers. Executing the hash function leads to inserting the flow identifiers to the hash table and creating appropriate lists for all slots showing a collision. 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:

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

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

and s is the number of memory slots. This function clearly penalizes candidate hash functions showing many collisions and thus long lists in the hash table with separate chaining. The following example demonstrates the fitness evaluation: Consider that two flow identifiers are assigned to slot $i = 3$, three input identifiers are assigned to slot $i = 10$ and 0 or 1 input is assigned to the remaining slots $(s = 20)$. Then $f(h) = 2^2 + (2^2 + 3^2) = 17$. The objective is to minimize $f(h)$.

Candidate programs usually contain redundant instructions. For example, they could contain instructions whose result is not used by any other instruction or whose execution does not affect contents of the registers. These instructions can be removed. As modern processors support SIMD (Single Instruction Multiple Data) processing via the SSE and AVX extensions, we re-arrange the candidate programs to fit this scheme [12]. For example, 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 eight instructions sequentially. The execution time of a candidate program then corresponds to the number of blocks of instructions, where one block contains all instructions that can be executed in parallel.

In a multi-objective scenario implemented by means of the NSGA-II algorithm [29], LGP thus tries to minimize the number of collisions and the number of instructions (or instruction blocks) [12].

D. Results

The network data used in experiments were collected with a network monitoring device installed in our research computer network. The 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.

Fig. 2 shows all Pareto fronts (the weighted number of collisions vs. the number of instructions in C code) obtained from 30 independent runs of LGP. We identified seven hash functions NSGAHash1 – NSGAHash7 covering the Pareto front for a further analysis.

Evolved hash functions and selected conventional hash functions were implemented in C and compiled with the

TABLE I
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
GPHash	2777	15052	48750
EFHash	5317	25266	63175
XORHash	2864	15011	**48575**
SipHash	2835	14934	48622
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
mixHash	2716	15006	48716

TABLE II
THE AVERAGE EXECUTION TIME ON INTEL XEON E5-2620V3

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
GPHash	1.448	4.749	12.406
EFHash	1.871	13.560	48.132
XORHash	0.649	2.390	6.774
SipHash	4.061	10.147	23.442
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
mixHash	0.566	2.178	6.352

identical compiler settings. These implementations were then used to evaluate the number of collisions (Table I) and CPU execution time (Table II) on test data sets.

Table II gives the average execution time needed to process all data sets 20 times. NSGAHash4 provides the shortest execution time because a good tradeoff between the number of collisions and the complexity of the hash function was discovered by LGP.

In summary, it was shown using real world network data that the proposed platform can provide high-quality compromise solutions (in terms of the execution time and the quality of hashing) in comparison with commonly used hash functions and specialized hash functions available in the literature.

IV. HASH FUNCTIONS IN FPGA

The hash functions evolved by LGP were optimized with respect to the number of collisions and the execution time on a CPU. LGP also tried to maximize the number instructions that can be executed in parallel. This property is useful from the hardware perspective as the evolved functions contain arithmetic operations that can be executed in parallel and the execution time can thus be minimized.

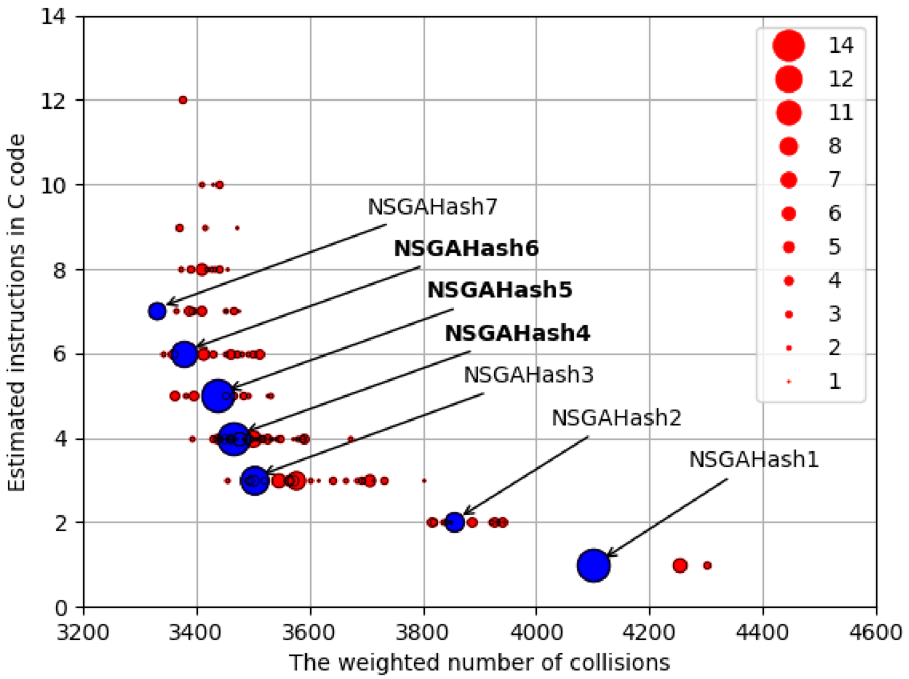
A. FPGA implementation

We analyzed evolved hash functions NSGAHash1 – NSGAHash7 and created their VHDL structural implementations according to evolved programs. In order to maximize the operation frequency, we inserted synchronization registers to enable the pipelined processing. Examples of resulting implementations are shown for NSGAHash4, NSGAHash5 and NSGAHash6 in Fig. 3, 4 and 5. The network flow description is provided in 32 bit registers i0, i1 and i2. Each stage of the pipeline contains a 32 bit function (such as addition, logic operation, rotation or no operation) followed by a 32 bit register R. Rotation is implemented by reconnecting the input signals according to a given bit count, i.e. no special function such as a barrel shifter is needed. The resulting 16 bit hash value is obtained from a 16 bit XOR function.

NSGAHash1 – NSGAHash7 were synthesized using Xillinx ISE 14.4 tool for three different Xilinx FPGAs, namely Spartan-6 (xc6slx150), Virtex-6 (xc6vlx550t) and Virtex-7 (xc7vx550t). Table III summarizes all important parameters: the latency, the number of look-up tables (LUTs), the number of flip-flops (FFs), delay and maximum operation frequency. In order to provide examples of conventional hash functions, we also implemented XORHash [24] and SipHash [25] and listed their parameters in Table III. It has to be noted that other conventional hash functions are more complex than the selected functions and their hardware implementation would not bring any advantages to our target application. One can observe that evolved hash functions are more compact than conventional hash functions (the number of LUTs and FFs was significantly reduced) and exhibit a small initial latency of 2–4 clock cycles. The execution time is comparable with XORHash, but SipHash is much slower than evolved hash functions.

B. Reconfigurable hash function

In order to design a reconfigurable hash function that could be used in the security use-case sketched in Section I, a natural solution would be to implement desired hash functions on the FPGA and select one of them by means of a multiplexer. Detailed analysis of NSGAHash4, NSGAHash5 and NSGAHash6 shown in Fig. 3, 4 and 5, however, revealed that these hash functions are structurally very similar. We took into account this fact and designed a new reconfigurable hash function (RecoHash) that contains all these hash functions. The multiplexers are carefully placed and used to switch among subcircuits of these hash functions rather than the whole hash functions. RecoHash has four different configurations,



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

  r[1] = rotr(r[1], 22);
  r[3] = r[2] + r[0];
  r[0] = r[1] + r[3];
  return r[0] ⊕ (r[0] >> 16);
}
unsigned int NSGAHash5 (input){
  r[0], r[1], r[2] = input;

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

Fig. 2. Resulting Pareto fronts created from 30 independent runs of LGP.

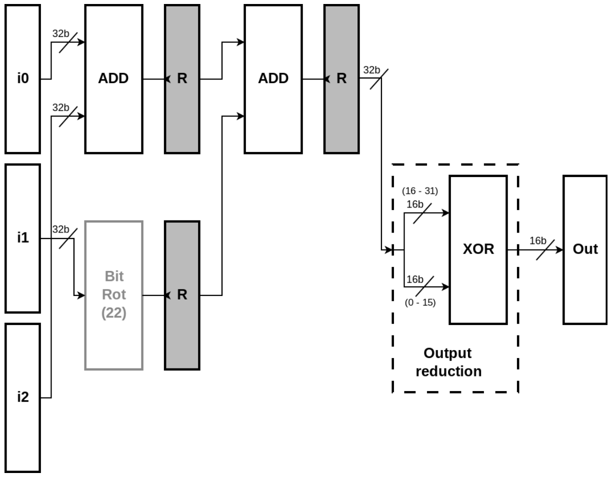


Fig. 3. Pipelined implementation of NSGAHash4.

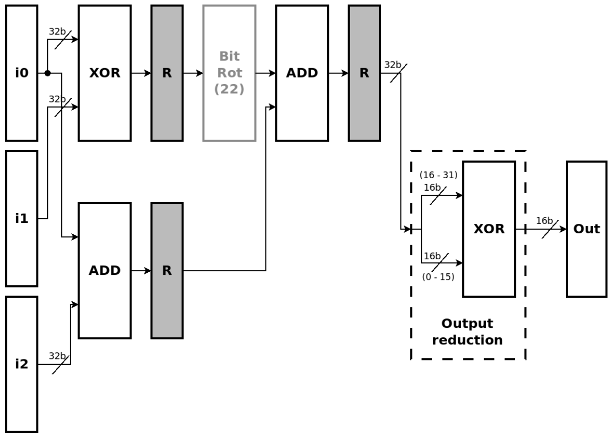


Fig. 4. Pipelined implementation of NSGAHash5.

implementing NSGAHash4 (mode 00), NSGAHash5 (mode 01), NSGAHash6 (mode 10) and mixHash (mode 11), where mixHash is a mixture of the former functions.

The synthesis results given in Table III clearly show the main benefits of RecoHash. For example, in the case of the implementation in Virtex-7, its size (144 LUTs) is significantly smaller than the sum of the LUTs needed to implement its core hash functions ($80 + 112 + 112 = 304$ LUTs). The same also holds for FFs ($144 < 3 \times 122$). The max. operation frequency of RecoHash is very close to the fastest hash functions.

Figure 7 shows all key parameters (LUTs, delay and the number of collisions) for all evolved hash functions, conventional hash functions and RecoHash. The number of collisions is given for the most challenging DataSet3. Because of the pipeline structure, evolved hash functions exhibit a very similar delay. The only exception is NSGAHash7 which is more complex due to multipliers that were implemented by DSP blocks available in the FPGA.

Finally, by means of 1 million input vectors we analyzed how many flows are hashed by RecoHash to the same index by means of its different configurations. As these numbers are

TABLE III

SYNTHESIS RESULTS FOR DIFFERENT TYPES OF XILLINX FPGAS. *NSGAHASH7 USES 3 DSP BLOCKS FOR MULTIPLICATION

Hash function	FPGA Type	Latency	Number of LUTs	Number of FFs	Delay [ns]	max. frequency [MHz]
SipHash	Spartan-6	4	989	521	10.501	95.23
	Virtex-6	4	1061	521	6.449	155.06
	Virtex-7	4	1061	521	5.469	182.84
XORHash	Spartan-6	7	291	228	2.395	417.54
	Virtex-6	7	291	228	1.771	564.65
	Virtex-7	7	291	228	1.594	627.35
NSGAHash1	Spartan-6	2	48	48	3.133	319.18
	Virtex-6	2	48	48	1.452	688.71
	Virtex-7	2	48	48	1.353	739.10
NSGAHash2	Spartan-6	3	80	112	2.358	424.09
	Virtex-6	3	80	112	1.766	566.25
	Virtex-7	3	80	112	1.589	629.33
NSGAHash3	Spartan-6	2	48	48	3.133	319.18
	Virtex-6	2	48	48	1.452	688.71
	Virtex-7	2	48	48	1.353	739.10
NSGAHash4	Spartan-6	3	80	112	3.133	319.18
	Virtex-6	3	80	112	1.766	566.25
	Virtex-7	3	80	112	1.589	629.33
NSGAHash5	Spartan-6	3	112	112	3.170	315.46
	Virtex-6	3	112	112	1.766	566.25
	Virtex-7	3	112	112	1.589	629.33
NSGAHash6	Spartan-6	3	112	112	3.170	315.46
	Virtex-6	3	112	112	1.766	566.25
	Virtex-7	3	112	112	1.589	629.33
NSGAHash7	Spartan-6	4	80*	161	11.541	86.65
	Virtex-6	4	80*	161	6.208	161.08
	Virtex-7	4	80*	161	5.432	184.09
RecoHash	Spartan-6	4	144	240	3.049	327.98
	Virtex-6	4	144	240	1.766	566.25
	Virtex-7	4	144	240	1.589	629.33

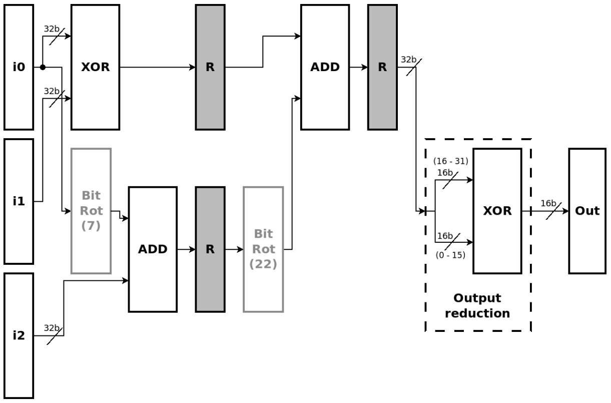


Fig. 5. Pipelined implementation of NSGAHash6.

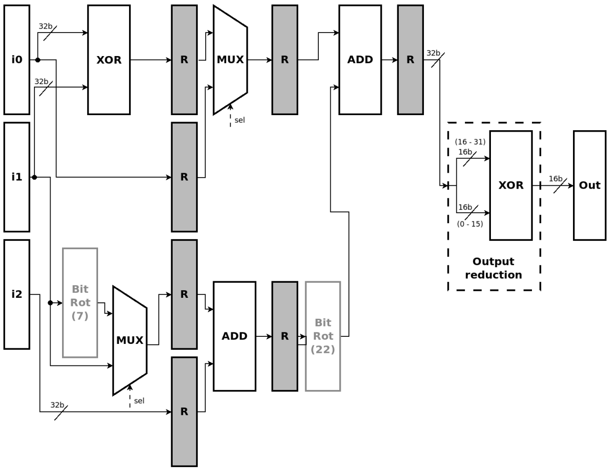


Fig. 6. Reconfigurable hash function RecoHash.

very low (e.g., 0.0021% for NSGAHash4 and NSGAHash5; 0.0017% for NSGAHash4 and NSGAHash6; and 0.0008% for NSGAHash5 and NSGAHash6), we concluded that RecoHash provides significantly different hash values in its operation modes.

V. CONCLUSIONS

Motivated by the recent need for the high-speed network flow processing in FPGAs, we proposed efficient hardware implementations of hash functions for an FPGA, including a reconfigurable hash function. The proposed solution exploits a multi-objective LGP capable of designing and optimizing not only the quality of hashing, but also the execution time of hash functions. Because of these properties, evolved hash functions (i.e. sequences of instructions) could directly be translated to a VHDL structural description, synthesized and evaluated on several FPGAs. Compared with conventional solutions, evolved implementations require less area in the FPGA while the maximum operation frequency is slightly higher.

We exploited the structural similarity of several hash functions and combined them together to create a reconfigurable

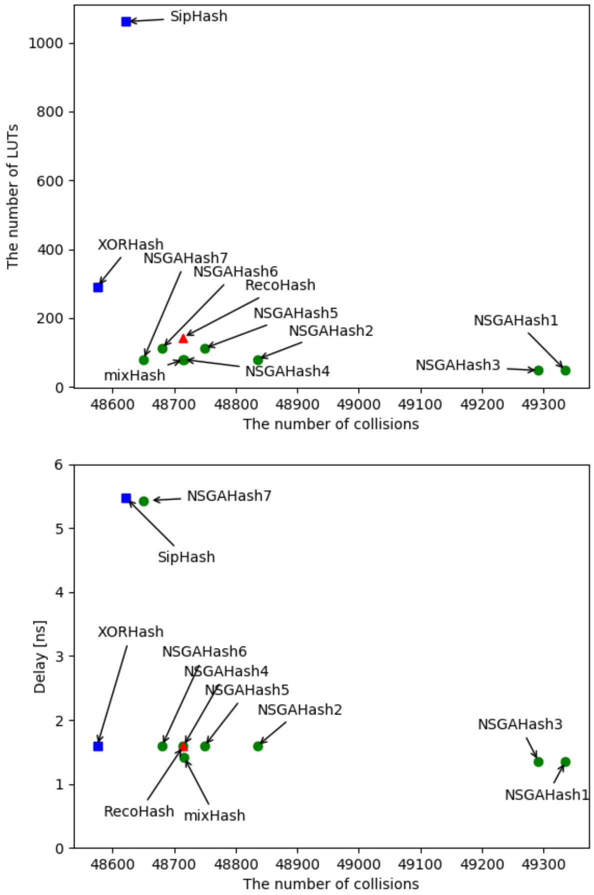


Fig. 7. Parameters of implementations of hash functions in Virtex-7. The number of collisions is given for DataSet3.

hash function RecoHash. RecoHash requires less area in the FPGA than any solution based on multiplexing of the available hash functions. RecoHash can also be used as a building block of more complex hashing schemes.

The quality of hashing was evaluated with the data coming from real network flows. In our future work, we plan to integrate selected hardware implementations of hash functions into SDM system and evaluate them in the real online scenario.

Acknowledgments

This work was supported by The Ministry of Education, Youth and Sports of the Czech Republic INTER-COST project LTC18053. The authors would like to thank Dr. Martin Zadnik for his valuable comments to this research.

References

[1] D. E. Knuth, *The Art of Computer Programming (Volume 3)*, 1973.

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

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

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

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

[6] A. Appleby, "Murmur hash functions," https://github.com/aappleby/smhasher.

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

[8] R. C. Merkle, "Secrecy, authentication, and public key systems," Ph.D. dissertation, Stanford University, 1979.

[9] M. Safdari and R. Joshi, "Evolving universal hash functions using genetic algorithms," in *In Proc. of the Future Computer and Communication*, 2009, pp. 84–87.

[10] C. Estebanez, Y. Saez, G. Recio, and P. Isasi, "Automatic design of noncryptographic hash functions using genetic programming," *Computational Intelligence*, vol. 30, no. 4, pp. 798–831, 2014.

[11] 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.

[12] ——, "Multiobjective evolution of hash functions for high speed networks," in *Proceedings of the 2017 IEEE Congress on Evolutionary Computation*. IEEE Computer Society, 2017, pp. 1533–1540.

[13] P. Berarducci, D. Jordan, D. Martin, and J. Seitzer, "Gevosh: Using grammatical evolution to generate hashing functions." in *MAICS*, 2004, pp. 31–39.

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

[15] 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.

[16] M. Kidoň and R. Dobai, "Evolutionary design of hash functions for ip address hashing using genetic programming," in *Evolutionary Computation (CEC), 2017 IEEE Congress on*. IEEE, 2017, pp. 1720–1727.

[17] Z. A. Kocsis, G. Neumann, J. Swan, M. G. Epitropakis, A. E. Brownlee, S. O. Haraldsson, and E. Bowles, "Repairing and optimizing hadoop hashcode implementations," in *International Symposium on Search Based Software Engineering*. Springer, 2014, pp. 259–264.

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

[19] 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.

[20] 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*, 2006, pp. 1861–1862.

[21] R. Salomon, H. Widiger, and A. Tockhorn, "Rapid evolution of time-efficient packet classifiers," in *2006 IEEE International Conference on Evolutionary Computation*, 2006, pp. 2793–2799.

[22] E. Damiani, A. G. B. Tettamanzi, and V. Liberali, "On-line evolution of fpga-based circuits: a case study on hash functions," in *Proc. of the First NASA/DoD Workshop on Evolvable Hardware*, 1999, pp. 26–33.

[23] R. Dobai, J. Korenek, and L. Sekanina, "Evolutionary design of hash function pairs for network filters," *Applied Soft Computing*, vol. 56, no. 7, pp. 173–181, 2017.

[24] 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.

[25] J.-P. Aumasson and D. J. Bernstein, "Siphash: A fast short-input PRF," in *Progress in Cryptology - INDOCRYPT 2012*. Springer, 2012, pp. 489–508.

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

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

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

[29] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.