BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

EVOLUTIONARY DESIGN AND OPTIMIZATION OF COMPONENTS USED IN HIGH-SPEED COMPUTER NETWORKS

EVOLUČNÍ NÁVRH A OPTIMALIZACE KOMPONENT POUŽÍVANÝCH VE VYSOKORYCHLOSTNÍCH POČÍTAČOVÝCH SÍTÍCH

PHD THESIS
DISERTAČNÍ PRÁCE

AUTHOR — **Ing. DAVID GROCHOL**
AUTOR PRÁCE

SUPERVISOR — **prof. Ing. LUKÁŠ SEKANINA, Ph.D.**
ŠKOLITEL

BRNO 2019

Abstract

The research presented in this thesis is directed toward the evolutionary optimization of selected components of network applications intended for high-speed network monitoring systems. The research started with a study of current network monitoring systems. As an experimental platform, the Software Defined Monitoring (SDM) system was chosen. Because traffic processing is an important part of all monitoring systems, it was analyzed in greater detail. For detailed studies conducted in this thesis, two components were selected: the classifier of application protocols and the hash functions for network flow processing. The evolutionary computing techniques were surveyed with the aim to optimize not only the quality of processing but also the execution time of evolved components. The single-objective and multi-objective versions of evolutionary algorithms were considered and compared.

A new approach to the application protocol classifier design was proposed. Accurate and relaxed versions of the classifier were optimized by means of Cartesian Genetic Programming (CGP). A significant reduction in Field-Programmable Gate Array (FPGA) resources and latency was reported.

Specialized, highly optimized network hash functions were evolved by parallel Linear Genetic Programming (LGP). These hash functions provide better functionality (in terms of quality of hashing and execution time) than the state-of-the-art hash functions. Using multi-objective LGP, we even improved the hash functions evolved with the single-objective LGP. Parallel pipelined hash functions were implemented in an FPGA and evaluated for purposes of network flow hashing. A new reconfigurable hash function was developed as a combination of selected evolved hash functions. Very competitive general-purpose hash functions were also evolved by means of multi-objective LGP and evaluated using representative data sets. The multi-objective approach produced slightly better solutions than the single-objective approach. We confirmed that common LGP and CGP implementations can be used for automated design and optimization of selected components; however, it is important to properly handle the multi-objective nature of the problem and accelerate time-critical operations of GP.

Keywords

Evolutionary algorithms, Cartesian Genetic Programming, Linear Genetic Programming, Network Monitoring, Network Application, Computer Network, Hash Function

Reference

GROCHOL, David. *Evolutionary design and optimization of components used in high-speed computer networks*. Brno, 2019. PhD thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Lukáš Sekanina, Ph.D.

Abstrakt

Výzkum prezentovaný v této práci je zaměřen na evoluční optimalizaci vybraných komponent síťových aplikací určených pro monitorovací systémy vysokorychlostních sítí. Práce začíná studiem současných monitorovacích systémů. Jako experimentální platforma byl zvolen systém SDM (Software Defined Monitoring). Detailně bylo analyzováno zpracování síťového provozu, protože tvoří důležitou součást všech monitorovacích systémů. Jako demonstrační komponenty pro aplikaci optimálních technik navržených v této práci byly zvoleny klasifikátor aplikačních protokolů a hashovací funkce pro síťové toky. Evoluční algoritmy byly zkoumány s ohledem nejen na optimalizaci kvality zpracování dat danou síťovou komponentou, ale i na čas potřebný pro výpočet dané komponenty. Byly zkoumány jednokriteriální i vícekriteriální varianty evolučních algoritmů.

Byl navržen nový přístup ke klasifikaci aplikačních protokolů. Přesná i aproximativní verze klasifikátoru byla optimalizována pomocí CGP (Kartézské Genetické Programování). Bylo dosaženo výrazné redukce zdrojů a zpoždění v FPGA (Programovatelné Logické Pole) oproti neoptimalizované verzi.

Speciální síťové hashovací funkce byly navrženy pomocí paralelní verze LGP (Lineární Genetické Programování). Tyto hashovací funkce vykazují lepší funkcionalitu oproti moderním hashovacím funkcím. S využitím vícekriteriální optimalizace byly vylepšeny výsledky původní jednokriteriální verze LGP. Paralelní zřetězené verze hashovacích funkcí byly implementovány v FPGA a vyhodnoceny za účelem hashování síťových toků. Nová rekonfigurovatelná hashovací funkce byla navržena jako kombinace vybraných hashovacích funkcí. Velmi konkurenceschopná obecná hashovací funkce byla rovněž navržena pomocí multikriteriální verze LGP a její funkčnosti byla ověřena na reálných datových sadách v provedených studiích. Vícekriteriální přístup produkuje mírně lepší řešení než jednokriteriální LGP. Také se potvrdilo, že obecné implementace LGP a CGP jsou použitelné pro automatizovaný návrh a optimalizaci vybraných síťových komponent. Je však důležité zvládnout vícekriteriální povahu problému a urychlit časově kritické operace GP.

Klíčová slova

Evoluční algoritmy, kartezské genetické programování, lineární genetické programování, monitorování síťového provozu, síťové aplikace, počítačové sítě, hashovací funkce

Citace

GROCHOL, David. *Evolutionary design and optimization of components used in high-speed computer networks*. Brno, 2019. PhD thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Lukáš Sekanina, Ph.D.

Evolutionary design and optimization of components used in high-speed computer networks

Prohlášení

Prohlašuji, že jsem tuto disertační práci vypracoval samostatně pod vedením pana prof. Ing. Lukáše Sekaniny, Ph.D. Další informace mi poskytli doc. Ing. Jan Kořenek, Ph.D. a Ing. Martin Žádník, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
David Grochol
September 5, 2019

Poděkování

Chtěl bych poděkovat svému školiteli Lukáši Sekaninovi za jeho odborné rady a pomoc při tvorbě této disertační práce. Dále bych chtěl poděkovat kolegům z ústavu za odborné konzultace. Poděkování rovněž patři rodině, kamarádům a známým za neutuchající podporu během celého studia.

Dále bych chtěl poděkovat projektům, v rámci kterých tato disertační práce vznikla:

- Pokročilé metody evolučního návrhu složitých číslicových obvodů, GA14-04197S
- Architektury paralelních a vestavěných počítačových systémů, FIT-S-14-2297
- Rozvoj kryptoanalytických metod prostřednictvím evolučních výpočtů, GA16-08565S
- IT4Innovations excellence in science, LQ1602
- Pokročilé paralelní a vestavěné počítačové systémy, FIT-S-17-3994
- Pokročilé metody nature-inspired optimalizačních algoritmů a HPC implementace pro řešení reálných aplikací, LTC18053

Contents

1 Introduction **3**
1.1 Research Objectives . . . 4
1.2 Thesis Outline . . . 5

2 State of the Art **6**
2.1 Computer Networks . . . 6
2.1.1 TCP/IP Model . . . 6
2.2 Network Monitoring . . . 8
2.2.1 Processing of Network Traffic and Network Flows . . . 10
2.2.2 FPGA Based Accelerators . . . 12
2.3 Hash Function Design . . . 14
2.4 Evolutionary Design . . . 16
2.4.1 Cartesian Genetic Programming . . . 18
2.4.2 Linear Genetic Programming . . . 19
2.4.3 Evolution of Hash Functions . . . 20
2.5 Multi-Objective EAs . . . 20

3 Research Summary **26**
3.1 Methodology . . . 26
3.1.1 The Use of Evolutionary Computation Methods . . . 26
3.1.2 Selection of Target Components . . . 27
3.1.3 Validation of Evolved Implementations . . . 27
3.2 Papers . . . 27
3.2.1 Paper I . . . 27
3.2.2 Paper II . . . 28
3.2.3 Paper III . . . 29
3.2.4 Paper IV . . . 30
3.2.5 Paper V . . . 31
3.3 List of Other Papers . . . 31

4 Discussion and Conclusions **33**
4.1 Contributions . . . 35
4.2 Future Work . . . 36

Bibliography **37**

Related Papers 45

I Evolutionary Circuit Design for Fast FPGA-Based Classification of Network Application Protocols 45

II Evolutionary Design of Fast High-quality Hash Functions for Network Applications 55

III Multiobjective Evolution of Hash Functions for High Speed Networks 64

IV Multi-Objective Evolution of Ultra-Fast General-Purpose Hash Functions 73

V Fast Reconfigurable Hash Functions for Network Flow Hashing in FPGAs 90

Chapter 1

Introduction

Many hardware providers have announced a support for 100 gigabit-per-second (Gbps) networks to overcome current 10–40 Gbps solutions [45, 57, 46]. The 400 Gbps and even 1 Tbps networks will be needed in a few next years, see Fig. 1.1. Commercial companies, data and supercomputer centers, and other entities around the world are now working towards launching 100 Gbps networks in order to benefit from faster communication and wider bandwidth in high-throughput requesting applications such as high-performance computing, high-quality video streaming or Internet of Things (IoT). Managing 100 Gbps networks 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 [43]. Having less than 7 ns to process one packet in a 100 Gbps network, SDM performs the analysis using relatively simple (and so fast) hardware whose functionality (i.e. the rules of operation) are defined in the software. Unrecognized traffic is then processed by sophisticated algorithms in the software. The analysis is performed at the level of *network flows*, where a network flow is defined as a set of packets (with the same key features) that passed an observation point in the network during a given time interval.

Because there are only a few nanoseconds to process each packet, monitoring systems have to be carefully designed and optimized. Traffic monitoring systems perform many operations with flows (such as an extraction of the information from packet headers and a deep packet inspection to determine the application protocol) As these operations have to be executed for each packet in the flow, it is important to provide their efficient (highly optimized) implementations in high-speed networks.

Evolutionary algorithms (EAs) have successfully been used to design and optimize many applications. The automated search for a new or an improved piece of software is a typical task specifically for genetic programming (GP). GP can be used to improve existing software (e.g. [55]), create or optimize parallel programs (e.g. [56]) or automate generating full computer programs (e.g. [4]). In recent years, significant development and progress have been reported in evolutionary circuit design. In many cases these techniques were capable of delivering efficient circuit designs in terms of an on-chip area minimization (e.g. [89]), adaptation (e.g. [40]), fabrication variability compensation (e.g. [90]), and many other properties (see, for example, many requirements on synthetic benchmark circuits in [84]).

The main focus of this thesis is the optimization of low-level HW/SW components of network monitoring systems. It is important to identify the components that significantly influence performance in currently developed and future implementations of these systems.

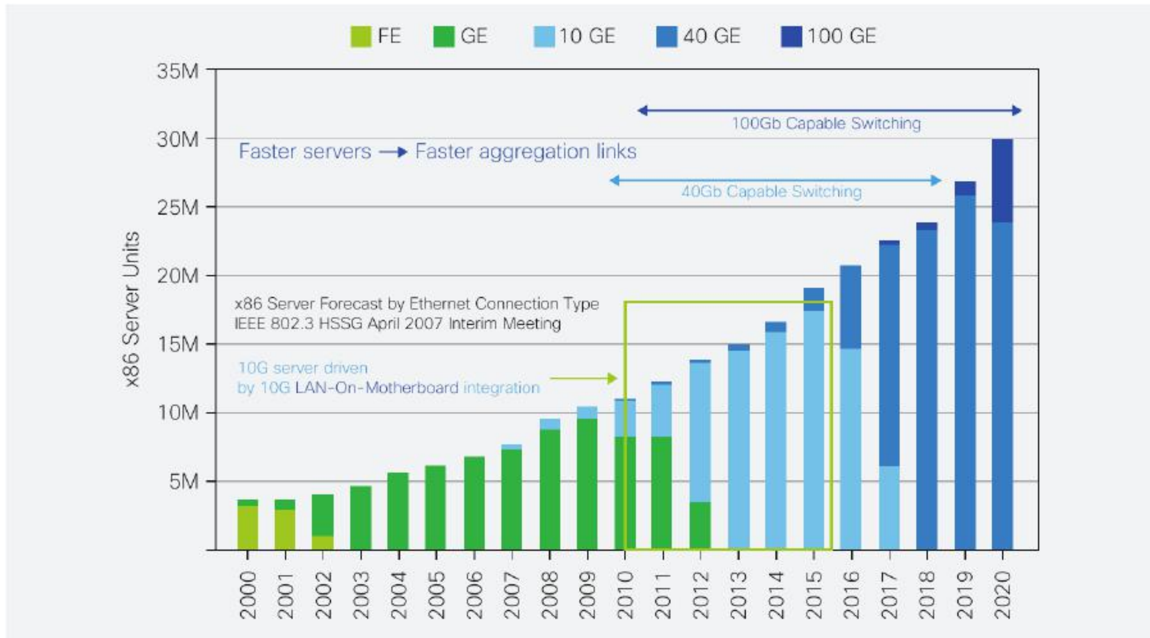


Figure 1.1: The network traffic is increasing rapidly in the last years (adopted from [11]).

The selected components require an optimization or a re-design not only in terms of functionality but also in terms of latency which is critical for high-speed networks. This thesis will explore how evolutionary algorithms (in particular genetic programming) can be employed to design and/or optimize selected components of high-speed network monitoring systems.

1.1 Research Objectives

We hypothesize that by using well-tuned evolutionary algorithms selected components of high-speed network monitoring systems can automatically be re-designed or optimized to improve their functionality and optimize other parameters (such as the implementation cost) in comparison with the state-of-the-art solutions.

The following research objectives were formulated:

1. To study network monitoring systems and identify their components suitable for automated optimization.

2. To define objectives and constrains that are important for an efficient optimization of the selected components.

3. To propose and implement single- and multi-objective variants of EAs including suitable fitness functions.

4. To validate the proposed approach and evolved solutions using relevant data sets.

5. To compare evolved solutions with the state-of-the-art implementations.

1.2 Thesis Outline

The thesis is composed as a collection of papers. The research contribution is presented in five peer-reviewed papers that are attached in section Related Papers. The thesis is organized as follows. Chapter 2 surveys the state-of-the-art. It primarily includes the principles of network monitoring and evolutionary design of circuits and programs. A special attention is devoted to the classification of application protocols in hardware and to the hash function design because these two problems are later selected as the case studies for this thesis. The research methodology and overview of scientific papers constituting this thesis is given in Chapter 3. Finally, Chapter 4 concludes the thesis and suggests possibilities for future research.

Chapter 2

State of the Art

This chapter provides a necessary background needed to understand the research presented in this thesis. Chapter 2.1 introduces relevant concepts of computer networks and network applications that are important for monitoring and security of computer networks. The hash function design is presented in Chapter 2.3. Chapter 2.4 surveys the principles of genetic programming including the graph-based and linear-based variants of GP. Multi-objective approaches for evolutionary algorithms are presented in Chapter 2.5.

2.1 Computer Networks

A computer network is a telecommunication network that provides connections among end-systems in order to share resources.

Computer networks [70] consist of end-systems (such as personal computers, servers, mobile devices or IoT nodes) and network devices (such as routers, switches, bridges, firewalls). These devices are connected by different types of links (wired, wireless or optical). Computer networks can be local, connecting nodes in the boundary of space, or global, connecting nodes all around the world (the Internet). The basic architecture of a computer network includes:

- A technology for signal transmission,
- a technology for reliable data transmission and
- an application layer which provides services for users.

Several network architecture models have been proposed. The reference network architecture model is the ISO/OSI model [97], which contains seven layers, namely Physical, Datalink, Network, Transport, Session, Presentation and Application layers. The most widespread model is the TCP/IP model, which is based on the ISO/OSI model, but uses a simpler architecture than the ISO/OSI model. It has only four instead of seven layers used in the ISO/OSI model. A comparison of these models is shown in Table 2.1.

2.1.1 TCP/IP Model

The TCP/IP model [33, 85] has been designed with respect to reliability, independence of the transmission medium, decentralized and simple implementation. Table 2.1 shows all the layers and the typical protocols associated with each layer. Each layer uses services

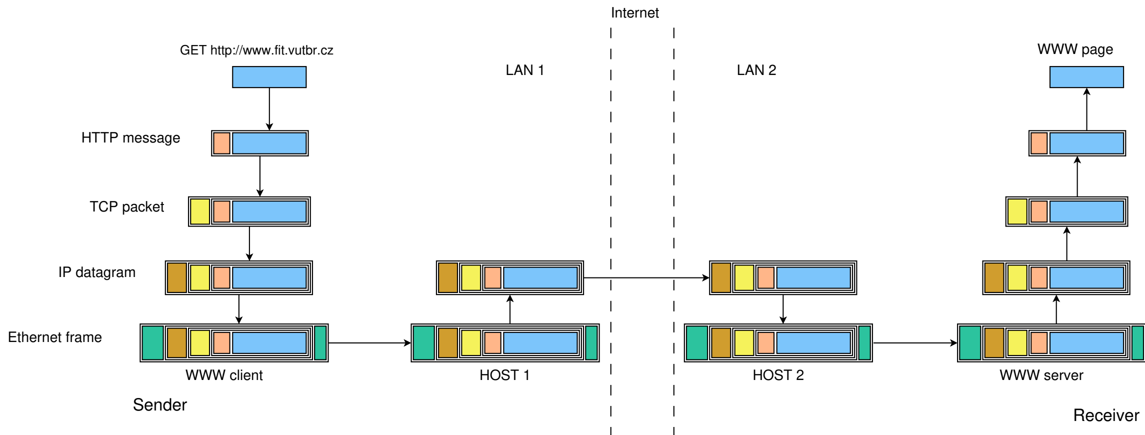


Figure 2.1: Example of the data encapsulation on the Sender side and the data decapsulation on the Receiver side.

provided by lower layers. Each layer adds a header or a footer to the data coming from the higher layers and delegates the encapsulated data to the lower layer for the next processing. Example of the data encapsulation on the sender side and decapsulation on the receiver side is shown in Figure 2.1. The lowest layer sends all the data with headers (footers) to a target device.

Network interface layer defines requirements on physical medium, electrical signals and optical signals. Examples of the network interface layer implementations are Ethernet, TokenRing, Frame Relay, FDDI or RS-232C. In addition to these requirements, the network interface layer defines the methods enabling the access to a physical medium.

Internet layer addresses and routes basic transfer structures (the so-called datagrams) containing the header and payload sections. Datagrams are routed using the best-effort delivery strategy, which tries to find a compromise between the shortest and the fastest paths through the entire network. If the datagram is lost during its transport via the network, the sender has to arrange for its re-sending. The internet layer typically uses five protocols. The Internet Protocol (IP) provides services for transport layer protocols. It distinguishes devices in the network. Two versions of IP are currently used IPv4 and IPv6, see Figure 2.2. Next protocols are Address Resolution Protocol (ARP) and Reverse ARP (RARP). The ARP translates an IP address to a MAC address and RARP translates a MAC address to an IP address. Internet Group Message Protocol (IGMP) is used for logging to multicast groups. Internet Control Message Protocol (ICMP) sends error messages in the networks, for example, a device goes offline or a service is unavailable. All internet layer protocols have to be implemented in the operation system.

Transport layer ensures logical connections between processes on the end devices connected through the IP. The logical connection is identified by a port number. The transport protocol divides the data coming from the application layer to smaller pieces and adds a header in order to form a packet. The packets create a sequence which is called the *network flow*. The network flow is defined as a set of packets with the same key features and is passed by an observation point in the network during a given time interval. The transport layer uses two protocols. (i) The Transmission Control Protocol (TCP) ensures reliable connections. It means that all the data sent from a sender will be delivered to a recipient correctly. The TCP adds special packets to the communication to establish the connection, finish the connection, confirm an acceptance, synchronize all entities, etc. (ii)

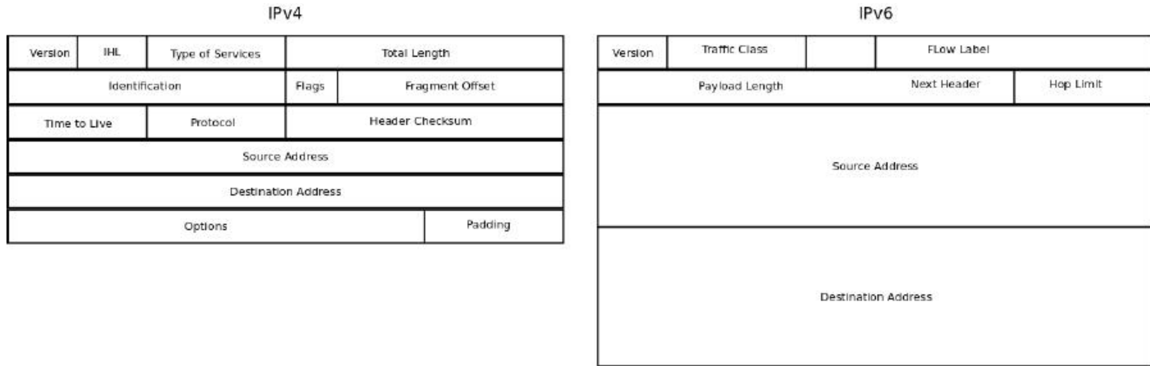


Figure 2.2: Comparison of IPv4 header (left) and IPv6 header (right).

Table 2.1: Comparison of ISO/OSI model and TCP/IP model.

ISO/OSI model	TCP/IP model	Group of Layers	Examples
Application Layer Presentation Layer Session Layer	Application Layer	Application Layer	Web Pages and Internet browsers
Transport Layer Network Layer	Transport Layer Network Layer	Internetwork Layer	TCP/IP Software
Data Link Layer Physical Layer	Data Link Layer Physical Layer	Hardware Layer	Ethernet ports, cables and ethernet drivers

The User Datagram Protocol (UDP), in fact, creates unreliable connections as there is no mechanism to check if a packet is delivered correctly. The data transfer is faster using UDP than TCP because there are no additional control packets. UDP is employed by services which need a fast transmission but can tolerate some undelivered packets, for example, Video on Demand (VOD), audio stream, etc.

Application layer covers many different application protocols developed for specific applications. Each protocol uses either TCP or UDP as transport protocol. Well-known protocols such as HTTP, SMTP, SIP, SSH have a port number assigned. The port numbers are divided into three ranges. The well-known ports, also known as the system ports, belong to the interval from 0 to 1023. These applications have to e strictly registered. The second group of ports (with numbers from 1024 to 49151) is a subject to registration at IANA organization. For the last group, with the so-called dynamic (or private) ports in the range from 49152 to 65535, there are no specific requirements for registration.

2.2 Network Monitoring

The network monitoring is crucial for ensuring the correct functionality of computer networks. It is based on probing of device states, traffic analysis and collecting traffic information. Results of monitoring are useful for administrators to improve network security, performance and functionality. Two types of network monitoring techniques exist [78, 66, 54]. (i) The *active monitoring* lies in injecting test traffic into the network and analyzing its impact. These tests can reveal real-time problems such as packet loss, jitter, insufficient bandwidth, unknown device status, latency and measure the quality of services (QoS). The

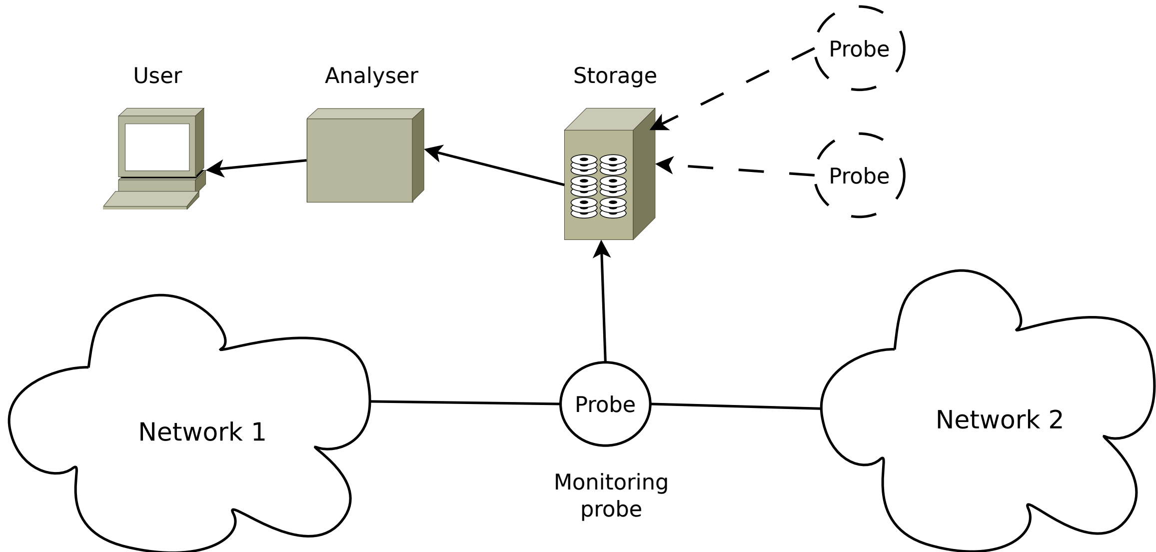


Figure 2.3: Schema of a monitoring system architecture.

active monitoring can thus relatively quickly detect network problems. The disadvantages of the active monitoring are increased network traffic and missing information about the data in the packed payload because only injected packets are analyzed. (ii) The *passive monitoring* is based on an analysis of real network traffic without injecting any new network traffic. It requires either special network devices to capture the network data or a built-in support in switches and other network devices. The passive monitoring provides more opportunities for the analysis than active monitoring. For example, the volume of the data generated by a device or anomalies in the traffic behavior can be detected. Any specific application, any user or any specific traffic may be observed and analyzed. The long-term statistics created from captured data are of a great importance for future infrastructure planning and upgrades. A monitoring probe is typically inserted between two networks, see Figure 2.3. The network probes typically send the data to a collector that analyses the data and distribute them to users or makes certain automatic or semi-automatic actions such as blocking of specific devices (users), applications or attacks in real-time.

Current high-speed networks operate at 10 Gbps and 40 Gbps. Solutions for 100 Gbps networks have been already demonstrated [43]. The specifications for 400 Gbps and even 1 Tbps are under design. High-speed networks require a novel approach to traffic monitoring because the monitoring systems used in 10 Gbps networks do not have enough throughput. A common feature of software implementations of network monitoring systems is that the monitoring probe is flexible and easily adapts to a given network. However, the software solution is often insufficient in terms of performance. Hardware implementations of monitoring systems, based on field-programmable gate arrays (FPGA) or application-specific integrated circuits (ASIC), usually have a sufficient throughput but it is often difficult to adapt them to a certain network or add new monitoring features.

A new approach to high-speed network monitoring known as Software Defined Monitoring (SDM) has recently been developed [43, 45]. The SDM combines hardware and software approaches. The SDM consists of three fundamental parts: i) a special hardware card with an FPGA based network monitoring accelerator, ii) a firmware controlling the data preprocessing in hardware and iii) user applications, see Fig. 2.4. The main benefit

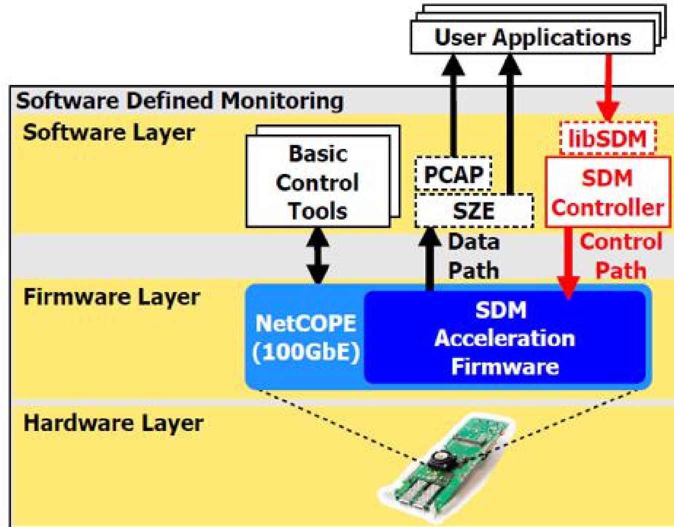


Figure 2.4: Software Defined Monitoring system architecture (adopted from [43]).

of SDM is its ability to control the hardware preprocessing of network flows by means of software applications. The control software sets the rules for the hardware accelerator with respect to the particular flows, groups of flows, devices (users) or selected protocols. It also sets various actions specifying how to discard packets, collect basic network characteristics, capture all packets for detailed analysis, etc. A rule or a set of rules is defined to preprocess each flow. Because the software part has to process only a small portion of the traffic (already preprocessed and aggregated data) it can be executed on a standard multi-core processor. About 80% of flows can be processed in hardware after the learning phase of the SDM system is finished [43]. However, during the learning phase, the software has to handle most of the flows.

2.2.1 Processing of Network Traffic and Network Flows

This section briefly describes the flow processing in the SDM system and identifies the most time-critical operations.

Network Traffic Processing

In the current networks, which are mostly based on TCP/IP model [72], a packet is formed by headers and payload. Every layer has its own header for identification and processing. A flow can be uniquely identified by a 5-tuple extracted from these headers: source and destination internet protocol addresses (IPv4 or IPv6), source and destination ports and transport protocol (mostly TCP or UDP). Each flow represents one direction of the communication between two applications on the devices. In the flow-based monitoring, we deal with the basic flow characteristics such as the flow length (the number of packets or bytes) and timing (the start time, the end time, the duration). These characteristics are often amended by certain interesting information from an application layer such as the type of protocol or some information extracted from the payload.

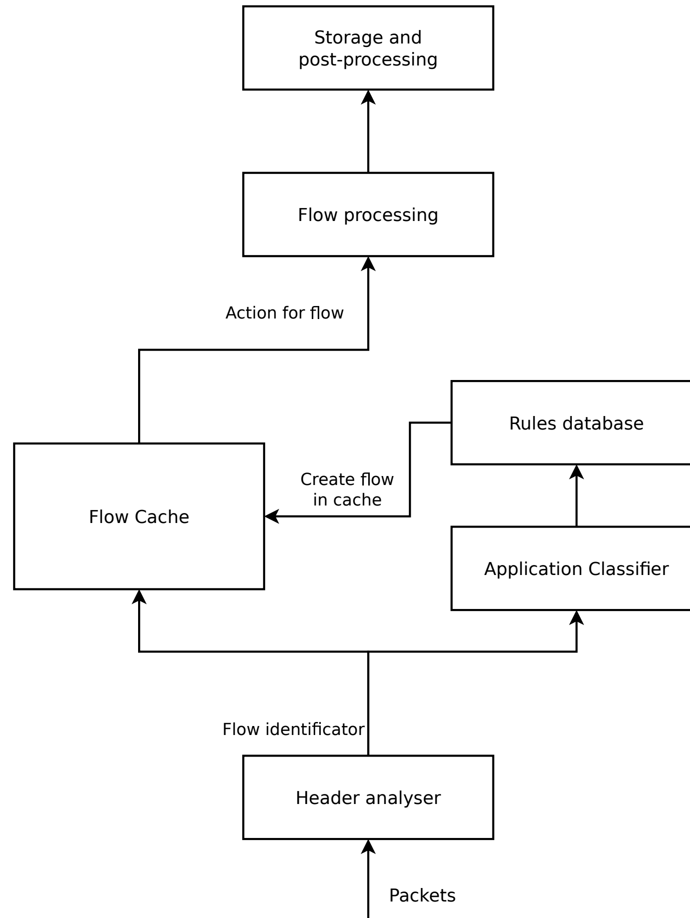


Figure 2.5: Packet processing in monitoring systems such as SDM.

Flow Processing

Fig. 2.5 shows a typical packet processing pipeline in the monitoring systems such as SDM. At first, the 5-tuple identification of the packet is extracted from packet headers. The packet is assigned to a flow record if the corresponding flow already exists in the flow cache. Otherwise, a new flow record is created in the flow cache and rules for the processing of the packets belonging to this flow are defined. There are different types of rules, for example, “capture all traffic”, “get basic characteristics” or “get advanced characteristics”. These rules can affect complete network traffic, a specific user, a subnet, a specific protocol or a group of protocols (e.g. communication protocols).

One of the challenges in network monitoring is the identification of application protocols. The research in the area of application identification has come up with distinct approaches to identify the applications carried out in the traffic. These approaches differ in the level of detail that is utilized in the identification method. The most abstract one is the *behavioral analysis* [38, 95]. Its idea is to observe only the port numbers and destination of the connections per each host and then to deduce the application running on the host by its typical connection signature. If more details per connection are available, the *statistical fingerprinting* [65] comes into play. In this case, a feature set is collected per each flow and the assumption is that the values of the feature set vary across applications, and hence, the applications leave a unique fingerprint. Behavioral and statistical fingerprinting

Table 2.2: Characteristics of different speed links. *Packet size is 64 bytes **CPU frequency = 3.6GHz

Link speed [Gbps]	Packets* per second	Time to process one packet [ns]	approximate CPU** clocks cycles
1	1 953 125	512.0	1843
10	19 531 250	51.2	184
40	78 125 000	12.8	46
100	195 312 500	5.12	18
400	781 250 000	1.28	5

generally classifies the traffic into the application classes rather than into the particular applications. The reason is that different applications performing the same task often exhibit similar behavior. For instance, application protocols such as Oscar (ICQ), MSN and XMPP (Jabber) transport interactive chat communications, and hence, they exhibit a similar behavior, which is very hard to differentiate for the monitoring system. The inability to distinguish applications within the same class is in some situations seen as a drawback, for example, when it is necessary to block/capture an application while other applications of the same class have to remain running. The approach utilizing the greatest level of detail is called *deep packet inspection*. It identifies applications based on the packet payload. The payload is matched with known patterns (defined, for example, by regular expressions) derived for each application [80].

The L7 filter [25] is a popular program for the application protocol identification, which utilizes regular expressions to describe the application protocols. It performs pattern matching in network flows. If a known pattern is matched in the payload, the corresponding application protocol is assigned to the network flow. Current processors are not powerful enough to achieve 100 Gbps throughput for the regular expression matching. The throughput of L7 decoder is less than 1 Gbps per one CPU core even for the latest Xeon processors [31, 32]. In order to achieve 100 Gbps throughput, it is necessary to use highly optimized hardware accelerators.

2.2.2 FPGA Based Accelerators

Table 2.2 shows how requirements on CPUs are growing with increased link speed. As a processor-based network monitoring is only applicable to 10 Gbps, hardware acceleration is needed for faster links.

Paxson et al. [69] argue that these performance requirements should be met by leveraging a high degree of possible parallelism that is inherent to network traffic monitoring. FPGAs, as well as ASICs, may deliver a vast support of parallelism. However, only FPGAs enable possibility to prototype and implement critical components for various network applications at the highest speeds while the optimized ASICs follow broad deployment a few years later. FPGAs are extensively used in the so-called hardware-accelerated network cards to implement the first line of network traffic processing such as monitoring, forwarding and other applications [1, 44].

FPGAs consist of programmable routing network and basic building blocks such as look-up tables (LUTs), registers, and block memories. A particular setup of the routing network defines the interconnection of these components. The LUTs serve to implement combinational logic while registers and block memories serve to keep the stateful information.

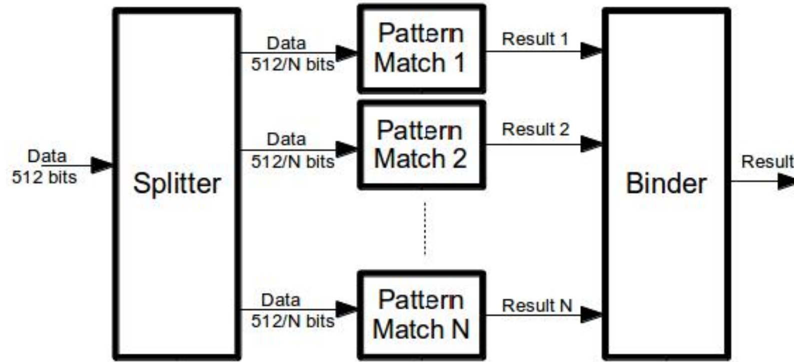


Figure 2.6: Increasing the throughput by multiple pattern matching units.

Modern FPGAs contain millions of LUTs and registers and thousands of block memories with the total capacity of hundreds MB [94]. All these components may, theoretical, work in parallel, independently of each other, and provide enormous computation power with low energy consumption in tens of Watts per chip. Moreover, FPGAs targeting the network market provide more than a hundred of high-speed transceivers allowing for connection to high-speed network links (e.g. high-end Virtex UltraScale+ FPGA offers up to 4 Tbps of aggregated transceiver throughput [93]).

The crucial task is to transform a high-level description of the circuit (for example, written in VHDL or SystemC) into an effective implementation in FPGA from the perspective of meeting the timing and resource constraints.

In recent years, many researchers have proposed high-speed pattern matching hardware architectures, which utilize the fine-grained parallelism of FPGA technology. Mapping of the regular expressions matching to an FPGA was first explored by Floyd and Ullman [26], who showed that pattern matching realized by a Nondeterministic Finite Automaton (NFA) can be implemented using a programmable logic array. Sindhu et al. [82] proposed an efficient mapping of NFAs to FPGA and Clark et al. improved the implementation by a shared decoder [12, 13] which significantly reduced the amount of consumed logic resources. The AMTH (At Most Two-Hot encoding) architecture [96] provides another improvement of the NFA implementation in the FPGA. The combination of one-hot and binary encoding reduces the amount of flip-flops, representing the NFA states.

Several authors introduced an optimized mapping of Perl Compatible Regular Expressions (PCRE), which are widely used in Intrusion Detection Systems (IDS), to the FPGA. Sourdis et al. published in [83] an architecture that allows for the sharing of character classes, static sub patterns and introduced components for efficient mapping of constrained repetitions to the FPGA. Lin et al. created an architecture for sharing infixes and suffixes [58]. Nevertheless, these optimizations are relevant only for large sets of PCREs in IDS systems.

The throughput of a pattern matching circuit is determined by the number of bytes processed within one clock cycle and frequency of the hardware matching unit. The FPGA technology limits the maximum frequency to several hundreds of MHz. To increase the processing speed, the NFA can be modified to process multiple bytes per one clock cycle [8]. Unfortunately, with the increasing size of the NFA input, the amount of NFA transitions grows exponentially. As a result, the hardware matching unit consumes more FPGA resources and its frequency decreases rapidly.

The throughput can be increased by introducing multiple parallel matching units. These units need additional logic resources and buffers to distribute the network data to the matching units and join the results. The overhead of parallel processing is illustrated in Fig. 2.6. First, the splitter has to assign the sequence number for every packet and store the packet to a buffer. The packet data are then sent with a lower rate to one of the parallel matching units. The units perform pattern matching and send the results to a binder, which contains buffers to put the results in the right sequence order.

Introducing the parallel matching units can improve the matching speed up to 100 Gbps, but only at the cost of significant overhead in terms of latency, FPGA logic resources and memory buffers. This overhead is avoided by focusing on highly optimized hardware architectures with high throughput and low latency [59, 60].

2.3 Hash Function Design

Hash functions are often employed in hardware accelerators of network monitoring systems. They are responsible for searching in the rule table, for distributing data to process units and for storing the flows data to database. For example, in the distribution unit, a hash function is called for each packet. In order to maximize the performance of network monitoring systems, hashing has to be not only of a high quality, but also fast.

A *hash function* is a mathematical function h that maps an input binary string (of length l_D) to a binary string of fixed length (l_R), $h : D \to R$, where $l_D > l_R$. The output value is called a hash value or simply hash [50]. The definition of hash function implies the existence of *collisions*, i.e. $h(d) = h(d')$, where $d, d' \in D$ are two different input messages. An important requirement imposed on hash functions is that a small change in the input should generate a large change in the output, which is called the *avalanche effect*. Good hash functions usually satisfy both criteria – maximizing the avalanche effect and minimizing the collision rate.

Two major types of hash functions exist, cryptographic and non-cryptographic. The cryptographic hash functions are suitable for cryptographic applications [86]. They have to satisfy many requirements, e.g.:

- *practical efficiency* – for $d \in D$ it is computationally efficient to find a hash value $r \in R$ s.t. $h(d) = r$;
- *first preimage resistance* (one-way) – for $r \in R$ it is computationally infeasible to find an input value $d \in D$ s.t. $h(d) = r$;
- *second preimage resistance* (weak collision resistance) – for $d \in D$ it is computationally infeasible to find a value $d' \in D$, s.t. $d' \neq d$ and $h(d') = h(d)$;
- *collision resistance* (strong collision resistance) – it is computationally infeasible to find two distinct values $d', d \in D$, s.t. $h(d') = h(d)$.

These requirements lead to more complicated construction of hash functions and, hence, the cryptographic hash functions need more time to compute the hash value than the non-cryptographic hash functions. The cryptographic hash functions have many applications, for example, in message authentication tools, digital signatures or in other forms of authentication.

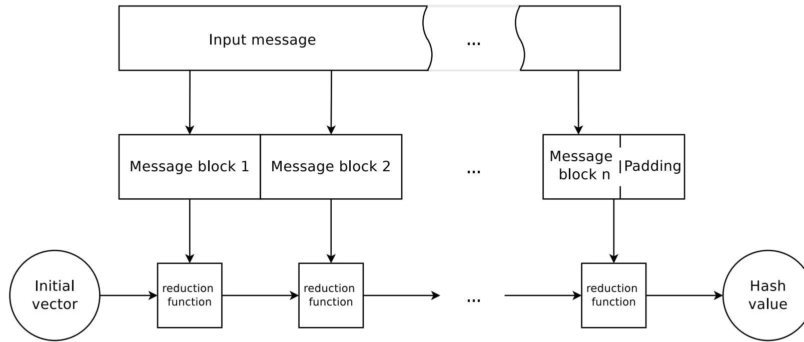


Figure 2.7: Merkle–Damgård construction of hash functions.

The non-cryptographic hash functions have to satisfy weaker requirements, but the practical efficiency and collision resistance are also important. These properties are often used to quantify the hashing quality of non-cryptographic hash functions.

Because the input size is usually arbitrary, hash functions are often designed using a pipelined (Merkle–Damgård) construction, see Fig. 2.7. It means that an input message is divided into blocks of a fixed size and processed block by block. The block is processed one at a time with an inside reduction function, each time combining the input block with the output of the previous round. The size of the output is typically the same as the size of the hash value. The last round produces the hash value. The last block of the message is typically padded with zeros to the required size.

The non-cryptographic hash functions have many applications, for example, in hash tables, search duplication, caches, bloom filters [61, 71, 36]. The *hash table* is a data structure used to implement an associative array, a structure which maps keys to values, see Figure 2.8. Hash tables have many applications, such as database indexing, object representation in programming languages or sets. Because hash functions produce collisions, it is necessary to resolve them in the hash tables. A well-known technique is *separate chaining*, where each slot in the hash table refers to a linear list that contains the records having the same hash. While determining the slot for a given input is performed in constant time, a particular record have to be searched sequentially. Next approach the collision

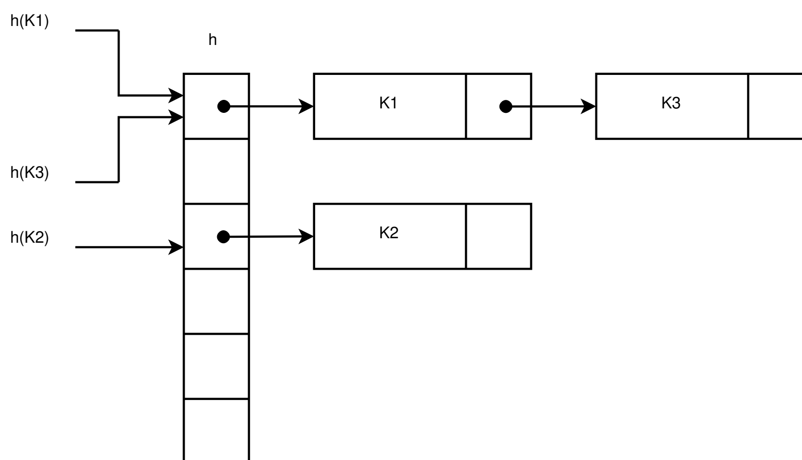


Figure 2.8: Example of hash table with size 6 slots, utilizing the separate chaining.

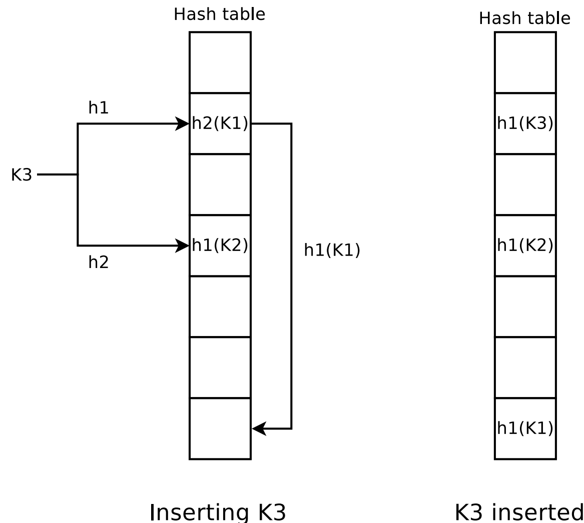


Figure 2.9: Example of inserting key to hash table with Cuckoo hashing.

resolving is the *open addressing*. This method searches (using some algorithms) for an alternative position in the hash table, where to store the data. Another approach, *cuckoo hashing* [68], uses two hash functions. A key is hashed by both hash functions and the data are stored to an empty slot indexed by one of them. If both slots are occupied, one of the keys stored in the table is rehashed by the other hash function and stored there, see example in Figure 2.9.

Many (non-cryptographic) hash functions have been proposed, for example, DJBHash [6], DEKHash [50], FVN (Fowler-Noll-Vo) [27], One At Time and Lookup3 [36]. MurmurHash2 and MurmurHash3, which are utilized in many open source projects, are hash functions suitable for general hash-based lookup [2]. CityHash is a family of non-cryptographic hash functions designed for fast hashing of strings [71]. Additional details are available in Paper II.

In addition to the general purpose non-cryptographic hash functions, there are also exist application-specific hash functions. They address specific properties of a particular application and, therefore, can be better (with respect to these properties) than the general-purpose hash functions. For hashing of network flows, the so-called XOR folding has been proposed [9]. Its implementation works with inputs of fixed size and is optimized in terms of performance.

SHMHasher [3] is a framework developed for evaluation of hash functions. It provides a test suite to evaluate the distribution, collision and performance properties of non-cryptographic hash functions. It contains many hash functions that can be used for a comparison. We used this framework in Paper V to measure the performance of hash functions.

2.4 Evolutionary Design

Evolutionary algorithms (EAs) [75] are inspired by the principles of biological evolution which is seen as an excellent optimization system. EAs are a class of stochastic optimization algorithms in which a population (a set) of candidate solutions is modified by genetic operations in order to solve a particular optimization problem. The quality of candidate

solutions is evaluated by means of the fitness function. A general evolutionary algorithm works as follows:

1. Initialize the population of candidate solutions (individuals).
2. Evaluate all individuals to determine their *fitness value*.
3. If termination conditions are met then stop. The result of EA is the individual with the best fitness value.
4. By means of a selection method select individuals from the population to a set of parents.
5. Create a set of offspring by applying genetic operators on the parents:
   (a) *Reproduction* – copy an individual to the offspring set unchanged
   (b) *Recombination* – exchange some parts of two or more individuals
   (c) *Mutation* – randomly modify some parts of an individual
6. Create a new population using the set of parents and offspring
7. Continue with step 2.

Many variants of evolution algorithms have been proposed in the literature, for example, evolution strategy [74], differential evolution [73], genetic algorithm [15] and genetic programming [52].

Genetic programming (GP) [52, 53] is primarily used for automated design of computer programs. Candidate programs are represented in memory as syntactic trees in the so-called tree version of GP. Nodes of the tree represent operations (arithmetic, logic, control etc.) and leaves contain terminal symbols such as program's inputs or constant values. During evolution, every candidate program is executed on a training data set in order to obtain its fitness value. Genetic operators randomly modify one candidate tree (mutation) or two or more candidate trees (swapping of subtrees) in crossover. The resulting tree is evaluated using a test set to validate its behavior on unseen data.

Other variants of GP use a different encoding of candidate programs. Cartesian GP and Linear GP are described in greater detail in the next chapter because they are relevant for this thesis.

2.4.1 Cartesian Genetic Programming

Cartesian genetic programming (CGP) has been developed by Miller since 1999 [64] and has been utilized in many applications as summarized in monograph [62]. A typical application of CGP is evolutionary circuit design. The idea of evolvable hardware and automated circuit design by means of artificial evolution was introduced by Higuchi et al. in 1993 [34]. A recent survey of the field covering key subfields (evolutionary hardware design and adaptive hardware) is available in [79]. In CGP, a candidate solution is modeled as a directed acyclic graph and represented in a 2D array of $n_c \times n_r$ processing nodes. Each node can perform one of the n_a-input functions specified in / set. The setting of n_c, n_r and / significantly influences the performance of CGP [63, 29].

The remaining parameters of CGP are the number of primary inputs (n_i), the number of primary outputs (n_o), and the level-back parameter (L) specifying which columns can

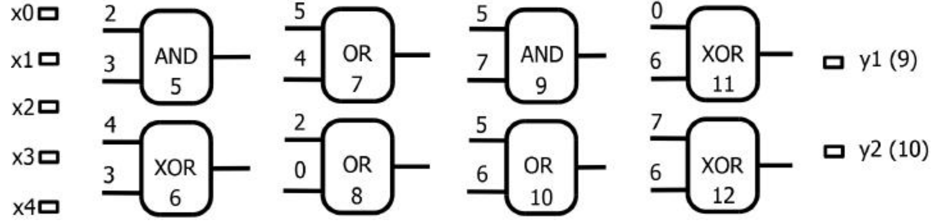


Figure 2.10: Example of a combinational circuit in CGP with parameters: $n_a = 2$, $n_i = 5$, $n_o = 2$, $L = 4$, $n_c = 4$, $n_r = 2$, $/ = \{$AND (0), OR (1), XOR (2)$\}$. Gates 8, 11 and 12 are not utilized. Chromosome: 2,3,0; 4,3,2; 5,4,1; 2,0,1; 5,7,0; 5,6,1; 0,6,2; 7,6,2; 9, 10. The last two integers indicate the outputs of the circuit.

be used as inputs for a given gate. The primary inputs are labeled $0 \dots n_i - 1$. The outputs of all nodes are labeled $n_i - 1 \dots n_c \cdot n_r + n_i - 1$ and considered as addresses where the connections can be fed to. In the chromosome, each n_a-input node is then encoded using $n_a + 1$ integers (n_a inputs and a node function). Finally, for each primary output, the chromosome contains one integer specifying the connection address. In CGP, the encoding is redundant because some nodes, some of their inputs or some primary inputs need not be used in the phenotype.

Algorithm 1: CGP

Input: CGP parameters, fitness function, original circuit p
Output: The highest scored individual and its fitness

1 $P \leftarrow$ CreateInitialPopulation(p);
2 EvaluatePopulation(P);
3 **while** ⟨*terminating condition not satisfied*⟩ **do**
4     $\alpha \leftarrow$ SelectHighest-scored-individual(P);
5   **if** *fitness*(α) $\geq$ *fitness*(p) **then**
6         $p \leftarrow \alpha$;
7     $P \leftarrow \{p\} \cup \{\lambda$ offspring of p created by mutation$\}$;
8   EvaluatePopulation(P);
9 **return** p, fitness(p);

CGP utilizes a search method known as $1 + \lambda$, where λ is the population size [62]. The initial population is randomly generated or seeded using conventional solutions. A new population consisting of λ individuals is generated by applying the mutation operator on the best individual of the previous population. The mutation operator randomly modifies h integers of the chromosome. The evolution is terminated after producing a given number of generations or a suitable solution is discovered, see Algorithm 1.

In the standard CGP used for combinational circuit evolution, the number of primary inputs n_i and outputs n_o is set accordingly to the requirements of the target circuit and / contains a set of Boolean functions. Figure 2.10 shows an example of a circuit and a corresponding chromosome.

A candidate circuit is evaluated by checking its responses for all possible input combinations. In order to accelerate the fitness function evaluation on a common processor, a bit-level parallel simulation of a candidate combinational circuit is employed. Contrasted

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

Figure 2.11: Example of a candidate program in LGP.

to a naïve simulation, in which 2^{n_i} vectors are sequentially submitted for evaluation (where n_i is the number of primary inputs), the bit-level parallel simulation exploits the fact that current processors enable performing bitwise operations over two w-bit operands in parallel [62]. Hence, the input vectors are grouped into w-bit words and simulated in parallel. The obtained speedup is w on a w-bit processor, for example, $64\times$ on a 64 bit common personal computer.

Although various new designs have been discovered using the standard CGP, the method is not directly applicable for the design of large combinational circuits because the fitness evaluation time grows exponentially with the number of primary inputs. Moreover, the number of requested fitness evaluations can easily go into millions, even for small (but nontrivial) circuits such as 4 bit multipliers. This problem has partially been eliminated by introducing circuit decomposition techniques at the representation level [87, 81] and formal verification methods in the fitness function [89]. Other successful applications of CGP have been proposed in domains in which candidate circuits are not evaluated using all possible input combinations (see, e.g., hash functions [41], image operators [88] or classifiers [40]).

The modern FPGAs contain 4- or 6-inputs LUTs. There are only a few papers dealing with the evolutionary circuit design at the level of 4-input LUTs [10, 41] and no paper dealing with 6-input LUTs. Unfortunately, the bit-level parallel simulation is inefficient for circuits consisting of LUTs because their logic function has to be emulated using a sequence of binary logic operations. As discussed in Paper I, employing CGP with 6-input LUTs (each of them encoded using 64 bits in the chromosome) would lead to long chromosomes, complex search spaces and very inefficient search procedures.

2.4.2 Linear Genetic Programming

Linear genetic programming (LGP) [7, 67, 92] is a variant of GP which uses a linear representation of candidate programs. Every program is composed of operations (called instructions) that are executed in a register machine. Operands, intermediate results and final results are stored in registers or in an external memory. Example of a candidate program is given in Figure 2.11. Linear GP evolves sequences of instructions in a machine language.

An instruction is typically represented by the instruction code, destination register and two source registers, for example, $[+, r0, r1, r2]$ represent the operation $r0 = r1 + r2$. The program result is returned in a predefined register. The number of instructions in a candidate program varies during the evolution, but the minimal and maximal size are defined. The number of registers available in the register machine is constant. The function

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

LGP is usually used with a tournament selection, one-point or two-point crossover and a mutation operator modifying either the instruction type or the register index. Advanced genetic operators have been proposed for LGP, for example [21, 22].

Like in other GP branches, the most computationally expensive part of LGP is the fitness function evaluation. In order to obtain the program's fitness score, the candidate program is executed on a set of training inputs, its outputs are collected and compared with desired values.

An individual can contain unused code parts, called *introns*, which do not affect the fitness value. However, the introns slow down the program execution. If introns are detected and eliminated, the evaluation time can be significantly reduced. According to [7], the existence of introns is important for the evolution process. Introns may act as a protection that reduces the effect of the variation process on the effective code.

The fitness function is typically focused on functionality, but other parameters of candidate programs can be optimized, such as the number of used instructions, execution time or power consumption of the processor.

2.4.3 Evolution of Hash Functions

Hash functions were successfully designed by evolutionary algorithms in recent years. The main advantage of EAs is that they are capable of producing high-quality hash functions optimized for a given application domain. Hash functions were evolved with genetic algorithms [76], tree GP [24], grammatical evolution [5] and Cartesian GP [91]. Both scenarios – application-specific hash functions (see, e.g., [42, 47, 51]) and general-purpose hash functions (see, e.g., [24, 39]) – were addressed in the literature. Relevant details are given in papers II, III, IV and V.

The fitness functions used in EAs developed for hash function design have been mostly focused only on the quality of hashing, usually expressed in terms of the collisions resistance, avalanche effect and distribution of outputs. The execution time of hash functions were not addressed by the GP literature before my research has been initiated.

2.5 Multi-Objective EAs

Previous chapters have dealt with single-objective EAs that produce solutions with respect to only one objective [18]. In many real-world problems such as the hash function design problem discussed in chapter 2.4.3, there are two or more optimization objectives that are conflicting. A simple approach is to combine several objectives into one (scaled) fitness function. Modern EAs, however, provide many useful techniques for truly multi-objective optimization [23, 16, 49].

A general multi-objective optimization problem is defined as follows:

Table 2.3: Solution relations in a multi-objective approach [23].

relation	notation	interpretation
strictly dominates	$x \prec\prec y$	$f_n(x) > f_n(y) \forall n$
dominates	$x \prec y$	$f_n(x) \geq f_n(y) \forall n \wedge \exists i : f_i(x) > f_i(y)$
weakly dominates	$x \preceq y$	$f_n(x) \geq f_n(y) \forall n$
incomparable	$x \parallel y$	$\neg(x \preceq y) \wedge \neg(y \preceq x)$
indifferent	$x \sim y$	$f_n(x) = f_n(y) \forall n$

$$\begin{gathered} \text{minimize/maximize } F(x) = (f_1(x), f_2(x), f_3(x), ...f_n(x)) \\ \text{subject to } g_i(x) \geq 0, i = 1, ..., m, \\ h_j(x) = 0, j = 1, ..., p, \end{gathered} \tag{2.1}$$

where f_i is the objective (fitness) function, n is the number of objectives and x is an individual. g_i and h_i are inequity and equity constraints, where m and p is the number of constrains. Various multi-objective algorithms have been proposed. These algorithms use different approaches to combine the optimization criteria and select new candidate solutions. A straightforward approach is to assign a weight for each fitness function. The final fitness function is the sum of the weighted fitness values:

$$fitness = \sum_{i=1}^{n} w_i f_i(x), \tag{2.2}$$

where w_i is the weight for i-th fitness function. Another approach is based on lexicographical sorting, in which individuals are gradually sorted by fitness values according to user preference. For example, VEGA algorithm [77] randomly divides the population into n subsets, where n is the number of objectives. Each subset is evaluated by one fitness function. The new population is formed by individuals from all subsets selected by a selection algorithm based on the fitness value.

The most successful multi-objective algorithms are based on the principle of *Pareto dominance*. We say that solution x Pareto dominates solution y if the following two conditions are fulfilled:

1. Solution x is better than solution y in at least one objective.
2. Solution x is no worse than solution y in all objectives ($x \prec y$).

This is formally captured by relation (the objective is to maximize F_i):

$$x \prec y : \forall n f_n(x) \geq f_n(y) \wedge \exists i : f_i(x) > f_i(y) \tag{2.3}$$

Table 2.3 summarizes all important relations between two solutions.

The set of solutions (out of all solutions) that are not dominated by any other solution forms *Pareto-optimal front* or *Pareto-optimal set*. Pareto front can also be constructed using solutions from a given population, i.e. using a subset of all possible solutions. Fig. 2.12 shows Pareto front (black) containing solutions which dominate the other solutions (white) in the population. The ultimate goal of the multi-objective algorithm is to find the Pareto-optimal set of solutions.

Evaluation algorithms utilizing the Pareto dominance employ different strategies to select individuals to the offspring population. Their main purpose is to maintain the diversity

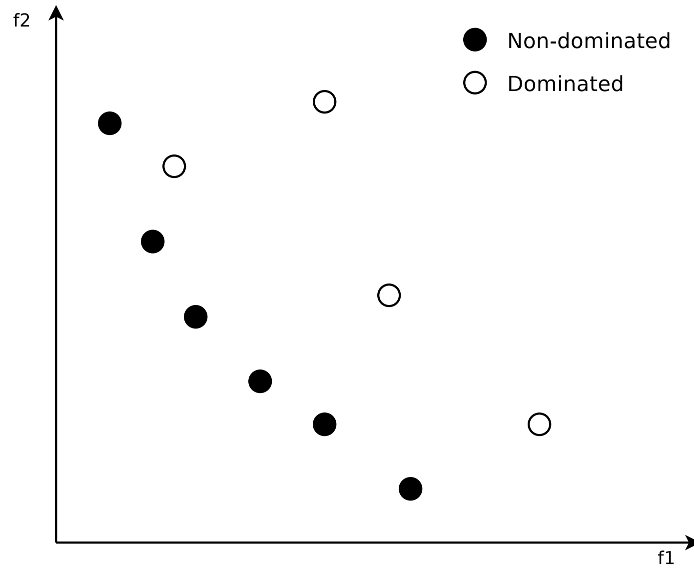


Figure 2.12: Individuals on the Pareto front (black points) dominate the remaining individuals (white points) in the population. f_1 and f_2 have to be minimized.

of the population. Selected multi-objective algorithms are described in detail in the next paragraphs.

Strength Pareto Evolutionary Algorithm 2

The Strength Pareto Evolutionary Algorithm 2 (SPEA2) was proposed by Zitzler et al. in 2001 [98, 48]. SPEA2 uses two sets of individuals (population and archive). The archive includes all non-dominated individuals from populations. If the archive is oversized, the number of individuals in the archive is reduced by the *truncation operation*; otherwise, if the archive is undersized, it is filled by the best dominated individual(s) from the population. The truncation operation performs the nearest neighbor algorithm on the individuals included in the archive. The individual with a minimal distance to another individual is chosen to be removed from the archive. The truncation operation removes individuals from the archive until the required size of the archive is reached.

SPEA2 uses a binary tournament selection with replacement, recombination and mutation for creating the offspring population.

Pareto Envelope-based Selection Algorithm II

Pareto Envelope-based Selection Algorithm II (PESA-II) [14, 28] employs a region-based selection, which enables to reduce the computational time for creating Pareto fronts. The search space is divided into hyper-boxes. Fitness functions assign every individual to a hyper-box. Using a standard selection method, a hyper-box is selected. Parents are randomly chosen from a given hyper-box and using standard genetic operators (crossover and mutation) offspring individuals are created. The selection algorithm operates with hyper-boxes instead of individuals.

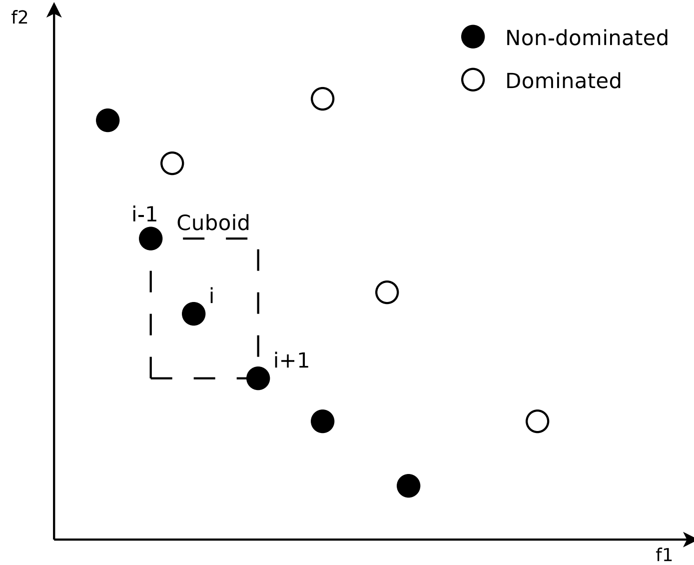


Figure 2.13: A cuboid used to determine the crowding distance of individual i.

Non-dominated Sorting Genetic Algorithm

One of the most popular multi-objective algorithms is Non-dominated Sorting Genetic Algorithm II (NSGA-II) proposed by Deb et al. in 2002 [17, 20, 37]. The algorithm is based on partitioning individuals from population P to non-dominated fronts. First front F_1 contains all non-dominated solutions. Every next front F_i is constructed as Pareto front for the population but individuals already included in F_{i-1}, F_{i-2}... are not considered. Each solution is assigned with a rank, which corresponds to the front ($p_{rank} = i$ for F_i). A naïve approach to create the non-dominated fronts requires $O(MN^3)$ operations, where M is the number of objectives and N is the population size.

The NSGA-II proposes a *fast-non-dominated sort*, see Algorithm 2, which requires $O(MN^2)$ operations. In this algorithm, the set S_p contains individuals from the population that are dominated by individual p. The number of individuals which dominate p is stored in n_p. Each individual p in the first front has $n_p = 0$. Creating next fronts is based on knowledge of S_p and n_p. For each solution in F_i, we visit each member q from S_p and

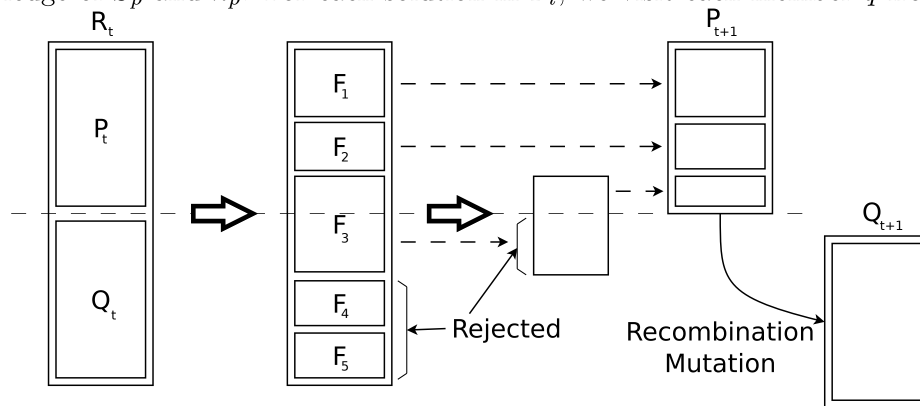


Figure 2.14: NSGA-II main algorithm step scheme.

Algorithm 2: Fast-Non-Dominated-Sort(P)

```
    Input: P - a population
    Output: F - a set of fronts of individuals
 1  F_1 = ∅;
 2  forall p ∈ P do
        /* Initialize all individuals in population */
 3      S_p = ∅;
 4      n_p = 0;
 5      forall q ∈ P do
            /* Compare all individuals */
 6          if p ≺ q then
 7              S_p = S_p ∪ {q};
 8          else if p ≻ q then
 9              n_p = n_p + 1;
        /* Create first front from non-dominated individuals */
10      if n_p = 0 then
11          p_rank = 1 ;
12          F_1 = F_1 ∪ {p};
13  i = 1;
    /* Create fronts of individuals.    */
14  while F_i ≠ ∅ do
15      Q = ∅;
16      forall p ∈ F_i do
            /* Remove individuals in front F_i from population and construct
               front from non-dominated individuals */
17          forall q ∈ S_p do
18              n_q = n_q − 1;
19              if n_q = 0 then
20                  q_rank = i + 1;
21                  Q = Q ∪ {q};
22      i = i + 1;
23      F_i = Q;
    /* Return constructed fronts of individuals */
24  F = (F_1, F_2...);
25  return F;
```

Algorithm 3: Crowding-Distance-Assignment(F_i)

Input: F_i - a set of individuals
Output: I - individuals with their crowding distance value

```
1 l = |F_i| ;
2 forall i ∈ F_i do
3     i_distance = 0;
4 forall m ∈ M do
5     I = sort(F_i, m);
6     I[1]_distance = I[l]_distance = ∞;
7     for i = 2 to (l − 1) do
8         I[i]_distance = I[i]_distance + (I[i+1]_m − I[i−1]_m) / (f_m^max − f_m^min);
9 return I;
```

decrement the domination value $n_q = n_q - 1$. If any member gets $n_q = 0$, we put it to the next front F_{i+1}. This process continues until all fronts are identified.

The crowding distance assignment algorithm (Algorithm 3), differentiates individuals inside a front. The algorithm estimates the perimeter of the cuboid formed by the nearest neighbors to determine the crowding distance $i_{distance}$, see Figure 2.13. The value $i_{distance}$ is the average side length of the cuboid. The algorithm requires to sort individuals for each objective value. The boundary solutions are assigned with an infinite distance value. Other solutions are represented by a normalized distance of two nearby individuals. The crowding distance is the sum of these values for each objective. The normalization is computed using f_m^{min} and f_m^{max} values, which are the minimum and maximum values of m-th objective function.

The main steps of NSGA-II algorithm are shown on Figure 2.14. Parent population P_t and offspring population Q_t, both of size N, are combined to auxiliary population R_t and sorted using the fast non-dominated sorting algorithm (Alg. 2). Each solution is assigned to a front. New parent population P_{t+1} is composed by adding individuals from fronts F_1, F_2... until the number of individuals in the population is N. When the number of individuals in P_{t+1} exceeds N, the crowding distance algorithm (Alg. 3) is used to select additional individuals according to the distance of the parent population P_{t+1}. An offspring population Q_{t+1} is created from P_{t+1} using standard genetic operation: selection, recombination and mutation.

Non-dominated Sorting Genetic Algorithm III (NSGA-III) is a new version of NSGA-II intended for many-objective problems [19, 35]. A many-objective problem has more objectives than a multi-objective problem, typically more than five. The main difference is in the selection algorithm, where it is important to maintain the diversity of the population. NSGA-III employs a different strategy for including candidate individuals to the new population.

Chapter 3

Research Summary

This chapter summarizes the research process conducted in order to write this thesis. Chapter 3.1 introduces the methodology adopted to fulfill the objectives specified in Chapter 1. Chapter 3.2 presents selected papers of the author, their abstracts and contributions. Chapter 3.3 lists other papers of the author that are not included in this thesis.

3.1 Methodology

The overall objective addressed in this thesis is to improve key parameters of selected components of high-speed network monitoring systems. Based on our survey of the state-of-the-art approaches to network monitoring reported in Chapter 2, SDM and its hardware/software implementation developed in [45] has been chosen as a framework suitable for arranging and performing our experiments.

3.1.1 The Use of Evolutionary Computation Methods

The proposed approach is based on employing well-known GP algorithms; LGP for program design and CGP for circuit design and optimization. As it is necessary to optimize not only one parameter of the components (for example, the quality of processing), the proposed design/optimization approach has to consider more design objectives. In the thesis, two approaches have been developed:

- a single-objective approach based on constraining one of the objectives (e.g. by specifying the maximum acceptable latency) and optimizing the other objective (e.g. the quality) and
- a truly multi-objective method optimizing all design objectives together.

In order to accelerate the design process, a parallel LGP implementation has been developed and evaluated. Moreover, as computing the exact values of some component's parameters (e.g. the exact program execution time) is relatively time consuming, we had to cheaply estimate these values to obtain the appropriate fitness value. Because evolutionary algorithms are non-deterministic heuristics, their outcomes have to be statistically analyzed and interpreted. It has systematically been done in all the case studies reported in the papers constituting this thesis.

3.1.2 Selection of Target Components

In order to evaluate the proposed EA-based design and optimization approach, we selected two components – a circuit implementing a simplified application protocol classifier and a software hash function. We believe that these two components provide many properties and features that characterize the type of design problems that have to be addressed in systems such as SDM. The most interesting features are:

- We do not usually know a "perfect implementation" of these components in terms of functionality. The design of these components is usually based on experimental work whose objective is to minimize an error metric on various data sets.
- In both cases, performance (in other words, delay) has to be optimized in addition to the functionality.
- One of the components (the classifier) is implemented as a digital circuit; the second component (the hash function) is primarily implemented as a software routine. There is thus an opportunity to investigate if, in principle, the same methodology can be applied for their design and what are the differences.
- Hash functions are developed as either application-specific or general-purpose functions. There is an opportunity to investigate if one evolutionary design method can lead to acceptable results for both the scenarios.

3.1.3 Validation of Evolved Implementations

EAs require some training data sets in order to establish a fitness value. Other data sets are needed to validate the evolved solutions. In both our case studies, we used real network data collected by co-authors of our papers to evaluate and validate the evolved solutions. We also used additional real world data and synthetic data to evolve general-purpose hash functions. We implemented state-of-the-art classifiers and hash functions to compare the results they produce with evolved solutions. In the case of circuit implementations we employed industrial design tools for FPGAs to obtain the area and delay of evolved circuits. In the case of hash functions, the execution time was measured on common processors.

3.2 Papers

This chapter presents the papers included in this thesis. For each paper, we present an abstract, a brief description with motivation and a summary of the main contributions. Full texts of all the papers are given in section Related Papers.

3.2.1 Paper I

GROCHOL David, SEKANINA Lukas, KORENEK Jan, ZADNIK Martin and KOSAR Vlastimil. Evolutionary Circuit Design for Fast FPGA-Based Classification of Network Application Protocols. Applied Soft Computing. Amsterdam: Elsevier Science, 2016, vol. 38, no. 1, pp. 933-941. ISSN 1568-4946.

Author participation: 40%
Journal Impact Factor (IF): 3.541

Abstract

The evolutionary design can produce fast and efficient implementations of digital circuits. It is shown in this paper how evolved circuits, optimized for the latency and area, can increase the throughput of a manually designed classifier of application protocols. The classifier is intended for high-speed networks operating at 100 Gbps. Because a very low latency is the main design constraint, the classifier is constructed as a combinational circuit in a field programmable gate array (FPGA). The classification is performed using the first packet carrying the application payload. The improvements in latency (and area) obtained by Cartesian genetic programming are validated using a professional FPGA design tool. The quality of classification is evaluated by means of real network data. All results are compared with commonly used classifiers based on regular expressions describing application protocols.

Contribution

In order to identify the application (or the application protocol) which the network traffic belongs to, one has to inspect one or several packets with a payload. The main difficulty is that the time to process one packet is less than 7 ns in the case of modern 100 Gbps links. This work is the extension of our initial work on the classifier design [30]. The main goal of the work is to show that these circuit classifiers can be optimized by means of Cartesian GP in order to reduce their latency and resources requirements. The improvements in latency and area obtained by CGP are validated by professional FPGA design tools. All results are compared with commonly used classifiers on several data sets.

This work introduces a new concept of the hardware classifier which is constructed as a fast-combinational circuit performing pattern matching over application protocols to be classified. We proposed accurate and relaxed versions of the classifier. Their circuit optimization by means of Cartesian GP led to 48.2% improvement in the area in FPGA (LUTs) and 19.8% improvement in latency with respect to an accurate human-designed classifier. Table 6 in Paper I shows results of synthesis for proposed classifiers. In order to compare the proposed solutions with the state-of-the-art classifiers from the literature, parameters of Yamagaki/Clark and AMTH circuit classifiers were included to this table. The classifiers were evaluated on real-network data.

3.2.2 Paper II

GROCHOL David and SEKANINA Lukas. Evolutionary Design of Fast High-quality Hash Functions for Network Applications. In: GECCO '16 Proceedings of the 2016 on Genetic and Evolutionary Computation Conference. New York, NY: Association for Computing Machinery, 2016, pp. 901-908. ISBN 978-1-4503-4206-3.

Author participation: 60%
Conference rank: A (Core)

Abstract

High-speed networks operating at 100 Gbps pose many challenges for hardware and software involved in the packet processing. As the time to process one packet is very short the corresponding operations have to be optimized in terms of the execution time. One of them is non-cryptographic hashing implemented in order to accelerate traffic flow identification.

In this paper, a method based on linear genetic programming is presented, which is capable of evolving high-quality hash functions primarily optimized for speed. Evolved hash functions are compared with conventional hash functions in terms of accuracy and execution time using real network data.

Contribution

One of the most frequently called functions in the flow processing is the hash function, which determines a memory address where the data of packet (flow) are stored. The goal of this work is to propose and evaluate a special hash function for flow hashing which has a good quality and is faster than the state-of-the-art hash functions. The hash function is constructed as a sequence of instructions for a CPU by means of a parallel linear GP exploiting the island model. In order to minimize the execution time, the hash function is constructed using a limited number of simple instructions. The evolved hash functions were compared with the hash functions available in the literature on real network datasets.

The paper shows that parallel single-objective LGP is capable of producing special hash functions for flow hashing. The program size is restricted to 12 instructions which was determined experimentally. Only simple instructions are used to minimize the execution time. The fitness function is based on the number of collisions and penalizing a solution generating many collisions on a given training data set. The evolved hash functions were compared with 11 hash functions available in the literature on real network data sets. The quality of hash functions is compared in Tab. 2 in Paper II. The best-evolved hash function has almost identical quality of hashing as the other hash functions but provides 3% improvement to the special network hash function (XORhash). Table 3 in Paper II compares the execution time of the hash functions on the CPU. The best-evolved hash function provides 26.9% improvement with the respect to the Murmur hash 3, which is typically used in SDM and which, on the other hand, provides a slightly lower number of collisions.

3.2.3 Paper III

GROCHOL David and SEKANINA Lukas. Multiobjective Evolution of Hash Functions for High Speed Networks. In: Proceedings of the 2017 IEEE Congress on Evolutionary Computation. San Sebastian: IEEE Computer Society, 2017, pp. 1533-1540. ISBN 978-1-5090-4600-3.

Author participation: 70%
Conference rank: B (Core)

Abstract

Hashing is a critical function in capturing and in an analysis of the 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.

Contribution

This work extends Paper II by including a multi-objective approach to the evolution process. The approach is based on the NSGA-II algorithm and linear GP. The multi-objective algorithm uses two fitness functions. The quality fitness function is taken from the previous work. The second fitness function estimates the execution time. Another contribution of this work is a new approach developed to quickly estimate the execution time of a candidate program. The execution time is estimated as a weighted number of instructions, where different weights are assigned to different types of instructions, based on their complexity. The estimation algorithm takes into account some features of modern CPUs, such as SIMD (Single Instruction Multiple Data) executions. The evolved hash functions were compared with hash functions available in the literature and hash functions obtained from our previous work.

This work resulted in an extension of LGP algorithm with a multi-objective approach. The quality of the execution time estimation is evaluated using randomly generated programs. The multi-objective method provided many non-dominated hash functions. Some of them are better than the commonly used hash functions and the specialized hash functions obtained by using the single-objective LGP with respect to chosen objective.

3.2.4 Paper IV

GROCHOL David and SEKANINA Lukas. Multi-Objective Evolution of Ultra-Fast General-Purpose Hash Functions. In: European Conference on Genetic Programming 2018. Berlin: Springer International Publishing, LNCS 10781, 2018, pp. 187-202. ISBN 978-3-319-77553-1.

Author participation: 70%
Conference rank: B (Core)

Abstract

Hashing is an important function in many applications such as hash tables, caches and Bloom filters. In the 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 a highly competitive quality of hashing but significantly reduced execution time in comparison with the state-of-the-art hash functions available in the literature.

Contribution

Paper II and Paper III have dealt with application-specific hash functions. This paper is focused on general-purpose hash functions that accept variable-length inputs, instead of a fixed-length input, which we considered in network hash functions. This change in the specification of hash functions led to the modification of the execution time estimation algorithm. Hence, the candidate (hash) program has to be wrapped to a loop, in which the input stream is processed block by block.

The evolved hash functions were compared with hash functions available in the literature on randomly generated data sets and real-world data sets (user passwords, network data,

Twitter and Facebook posts). The evolved hash functions produce a very similar number of collisions as other good hash functions from the literature on all data sets. However, evolved hash functions exhibit the shortest execution time in almost all cases on randomly generated and real-world data sets. They are slower than the special network hash functions, but faster than the general purpose hash functions when evaluated on the specific network datasets.

3.2.5 Paper V

GROCHOL David and SEKANINA Lukas. Fast Reconfigurable Hash Functions for Network Flow Hashing in FPGAs. In: Proceedings of the 2018 NASA/ESA Conference on Adaptive Hardware and Systems. Edinburgh: Institute of Electrical and Electronics Engineers, 2018, pp. 257-263. ISBN 978-1-5386-7753-7.

Author participation: 67%
Conference rank: unknown

Abstract

Efficient monitoring of high-speed computer networks operating with a 100 Gbps data throughput requires a suitable hardware acceleration of its key components. We present a platform capable of automated design 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 FPGA. Several evolved hash functions were combined together in order to form the 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. The characteristics of evolved hash functions were compared with the state-of-the-art hash functions in terms of the quality of hashing, chip area and the operation frequency in the FPGA.

Contribution

Using the methodology developed in Paper II and Paper III, we evolved hash functions suitable for FPGA implementations. We also introduced reconfigurable hash functions.

The evolved hash functions were translated to VHDL. In order to maximize their throughput, we added synchronization registers to enable pipelined processing. One of the reconfigurable hash functions was constructed using three evolved hash functions. These hash functions employ similar basic components that can be shared in the FPGA. The proposed reconfigurable hash function thus needs less than 50 % resources in comparison with the sum of resources needed to independently implement the three original hash functions.

3.3 List of Other Papers

- GROCHOL David, SEKANINA Lukas, ŽÁDNÍK Martin and KOŘENEK Jan. A Fast FPGA-Based Classification of Application Protocols Optimized Using Cartesian GP. In: Applications of Evolutionary Computation. Berlin: Springer International Publishing, LNCS 9028 , 2015, pp. 67-78. ISBN 978-3-319-16548-6.

Author participation: 50%

Conference rank: unknown

- GROCHOL David. Evoluční hardware v síťových aplikacích. In: Počítačové architektury a diagnostika PAD 2016. Bořetice: Faculty of Information Technology BUT, 2016, pp. 57-60. ISBN 978-80-214-5376-0.

  Author participation: 100%

  Conference rank: unknown

- GROCHOL David and SEKANINA Lukas. Comparison of Parallel Linear Genetic Programming Implementations. In: Recent Advances in Soft Computing: Proceedings of the 22nd International Conference on Soft Computing (MENDEL 2016) held in Brno, Czech Republic, at June 8-10, 2016. Cham: Springer International Publishing, 2017, pp. 64-76. ISBN 978-3-319-58088-3.

  Author participation: 60%

  Conference rank: unknown

Chapter 4

Discussion and Conclusions

This chapter summarizes the results presented in this thesis and outlines some possibilities for a future research.

The research presented in this thesis was directed toward the optimization of selected components of network applications intended for high-speed network monitoring systems. The work started with a study of current network monitoring systems. As an experimental platform, the SDM system was chosen. Because the traffic processing is an important part of all monitoring systems, it was analyzed in a greater detail. For detailed studies conducted in this thesis two applications were selected: the classifier of application protocols and the hash functions for flow processing. The evolutionary computing techniques were surveyed with the aim to optimize not only the quality of processing, but also the execution time. The single-objective and multi-objective versions of evolution algorithms were considered. The gained knowledge was summarized in Chapter 2 and used as background for the following research.

The research started with the design and optimization of the application protocol classifier. As the SDM required an accurate classification of application protocols, the classifier was based on deep packet inspection (by means of the application data). The proposed application protocol classification is based on a pattern matching algorithm, which is a time-consuming operation, emphasizing the need for a hardware acceleration. The current approaches require a lot of resources in hardware. A new approach was proposed to classify a small set of protocols in the FPGA (denoted CL-acc in Paper I). The classifier was synthesized by a professional design tool to an FPGA, see Table 4.1. The final circuit of the classifier was optimized using CGP (in Table 4.1 denoted as +CGP), reducing thus the amount of resources and latency. We also proposed relaxed implementations of the classifier. CL-cmp is a compromised version of the classifier (showing an additional area reduction for a small error in classification) and CL-lat is a minimal version of the classifier. Both relaxed classifiers were optimized by CGP. which enabled us to achieve a significant reduction of resources and latency. Table 4.1 also shows parameters of the state-of-the-art classifiers based on finite state machines (Yamagaki/Clark and AMTH). It can be seen that the proposed classifiers exhibit significantly better parameters. The accuracy of the classifiers was verified on real network data. On the other hand, FSM-based classifiers are more flexible and scalable.

The research continued with the hash function design using LGP. The first specialized network hash functions (evaluated for flow hashing) were optimized for the quality of hashing and constructed using a limited number of simple instructions. Single-objective and multi-objective LPG implementations were proposed for this purpose. Using the multi-

Table 4.1: Results of classifiers synthesis for the Xilinx Virtex-7 XC7VH580T FPGA (taken from Paper I, Table 6).

Classifier	LUTs	Flip Flop	Latency [ns]
CL-acc	2352	0	6.410
CL-acc+CGP	1909	0	6.113
CL-cmp	1549	0	6.093
CL-cmp+CGP	1073	0	5.604
CL-lat	1625	0	5.943
CL-lat+CGP	1217	0	5.139
Yamagaki/Clark	10431	2326	77.504 (16 x 4.844)
AMTH	10547	2190	71.536 (16 x 4.671)

Table 4.2: The average execution time on CPU for network data sets and real-world data sets (taken from Paper V, Table 7 and 8).

Hash function	Time [ms]					
	NetSet1	NetSet2	NetSet3	Passwords	Facebook	Twitter
DJBHash	1.861	5.134	12.724	5438.594	17.331	16.726
DEKHash	1.221	4.373	10.407	5067.882	13.240	13.119
FVNHash	1.301	4.721	9.633	5499.328	14.174	12.767
One At Time	1.769	5.290	12.352	6072.904	15.410	13.955
lookup3	0.925	2.891	7.435	4543.399	12.009	10.919
Murmur2	1.034	3.095	7.925	4464.339	11.723	10.774
Murmur3	1.193	3.215	8.727	4573.453	11.955	10.966
CityHash	0.960	2.625	7.407	4385.625	11.149	10.355
XORHash	0.838	2.318	6.652			
GPHash	1.865	4.671	12.558	6389.323	17.966	16.167
EFHash	2.472	13.527	49.495	5101.523	14.304	13.746
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	**4268.402**	***10.895***	***9.996***
EvoHash2	0.830	2.825	7.835	***4277.341***	**10.832**	**9.954**

objective approach, hash functions (NSGAHash1, NSGAHash2, NSGAHash3, NSGAHash4, NSGAHash5, NSGAHash6, NSGAHash7) were evolved, showing a better trade-off between the quality of hashing and the execution time than the state-of-the-art hash functions. The pipelined versions of network hash functions were implemented for FPGA. An adaptive configurable hash function was also created from three evolved hash functions. Several high-quality general-purpose hash functions (EvoHash1, EvoHash2) were also evolved using the proposed method.

All evolved hash functions were evaluated on real-world network data sets (see NetSet1, NetSet2, NetSet3 in Paper V) and common real-world data sets (Passwords, Facebook,

Twitter), see Table 4.3. The general-purpose hash functions were further evaluated on social network and randomly generated data sets. The evolved hash functions exhibit the same or better quality of hashing, but provide shorter execution time than the state-of-the-art hash functions.

4.1 Contributions

The section summarizes main contributions presented in this thesis, with respect to the research objectives formulated in Chapter 1.1:

Classification of application protocols:

- A new approach to the application protocol classifier design was proposed. Accurate and relaxed versions of the classifier were optimized by means of CGP. A significant reduction in FPGA resources and latency was reported in Paper I. A possible disadvantages of the proposed approach is that common classifiers are more flexible and scalable.

Hash Functions:

- Specialized, highly optimized network hash functions were evolved by parallel LGP. These hash functions provide better functionality (in terms of quality of hashing and execution time) than the state-of-the-art hash functions (Paper II).

Table 4.3: The number of collisions for network data sets and real-world data sets (taken from Paper V, Table 4 and 5).

Hash function	The number of collisions					
	NetSet1	NetSet2	NetSet3	Passwords	Facebook	Twitter
DJBHash	2835	15113	48925	11663	247	137
DEKHash	2926	15247	49017	14114	357	153
FVNHash	2756	14957	48780	11845	115	115
One At Time	2821	14988	**48636**	11590	105	138
lookup3	2742	15009	48737	11567	119	107
Murmur2	2800	15050	48749	11637	112	123
Murmur3	2744	***14911***	48763	11589	103	***89***
CityHash	2807	14990	***48647***	***11530***	122	122
XORHash	2864	15011	48575			
GPHash	2777	15052	48750	11634	117	113
EFHash	5317	25266	63175	983806	873270	824153
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	11871	***23***	98
EvoHash2	2821	14982	48695	**11469**	**10**	**1**

- Using the multi-objective LGP, we evolved a set of non-dominated hash functions showing better trade-offs between the quality of network flow hashing and the execution time in comparison with the state-of-the-art hash functions (Paper III).
- Parallel pipelined hash functions were implemented in an FPGA and evaluated for purposes network flow hashing. A new reconfigurable hash function was developed as a combination of selected evolved hash functions (Paper V).
- Very competitive general-purpose hash functions were evolved by means of the multi-objective LGP and evaluated using representative data sets (Paper IV).

We also confirmed that common LGP a CGP implementations can be used for automated design and optimization of selected components; however, it is important to:

- properly handle the multi-objective nature of the problem and
- accelerate time-critical operations (particularly the fitness calculation).

Based on these results, it can be concluded that the initial hypothesis of this research has been confirmed. The proposed EAs can design and optimize selected components of network applications of high-speed network monitoring systems and improve their key parameters.

4.2 Future Work

Based on our experience gained during this research the following future research directions were identified:

- Network monitoring systems are large and complex systems composed of many components. It is not a straightforward task to identify the critical components that should be optimized. Automated identification of such components for evolutionary re-design/optimization would be of high importance.
- An automated runtime optimization of components would be useful because the monitoring system could be adapted to the actual state of the system. If the optimization is fast, the component can be optimized for a specific situation or an important subset of input data.
- If the automated identification of components in network monitoring systems is connected to the runtime optimization, the system could adapt different components in runtime in a variable environment.
- Modern CPUs utilize many complex instructions, including application-specific instructions, such as hash function (for example Intel CPU: SHA1RNDS4 or AESDECLAST), special floating-point instructions or SIMD instructions (MMX and SSE). A future research could be focused on identifying a suitable subset of instructions that can be utilized by LGP; considering all possible instructions in LGP seems to be intractable.
- Other HW parts of monitoring systems those implemented in FPGA can be optimized using a multi-objective CGP. Contrasted to our work based on gate-level circuit optimization, a future work could deal with LUT-based circuit optimization in order to obtain more efficient FPGA implementations.

Bibliography

[1] Antichi, G.; Giordano, S.; Miller, D.; et al.: Enabling open-source high speed network monitoring on NetFPGA. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*. April 2012. pp. 1029–1035.

[2] Appleby, A.: Murmur hash functions. Https://github.com/aappleby/smhasher, [ONLINE, accessed: 31. 1. 2016].

[3] Appleby, A.: SMHasher. Https://github.com/aappleby/smhasher, [ONLINE, accessed: 1. 11. 2017].

[4] Becker, K.; Gottschlich, J.: AI Programmer: Autonomously Creating Software Programs Using Genetic Algorithms. *arXiv preprint arXiv:1709.05703*. 2017.

[5] Berarducci, P.; Jordan, D.; Martin, D.; et al.: GEVOSH: Using Grammatical Evolution to Generate Hashing Functions. In *MAICS*. 2004. pp. 31–39.

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

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

[8] Brodie, B. C.; Taylor, D. E.; Cytron, R. K.: A Scalable Architecture For High-Throughput Regular Expression Pattern Matching. *SIGARCH Computer Architecture News*. vol. 34, no. 2. 2006: pp. 191–202. ISSN 0163-5964.

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

[10] Cheang, S. M.; Lee, K. H.; Leung, K. S.: Applying Genetic Parallel Programming to Synthesize Combinational Logic Circuits. *IEEE Transactions on Evolutionary Computation*. vol. 11, no. 4. 2007: pp. 503–520.

[11] Cisco: The Future Is 40 Gigabit Ethernet. 2016. c11-737238-00.

[12] Clark, C.; Schimmel, D.: Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In *Field Programmable Logic and Application, 13th International Conference*. Lisbon, Portugal. 2003. ISBN 3-540-40822-3. pp. 956–959.

[13] Clark, C. R.; Schimmel, D. E.: Scalable Pattern Matching for High-Speed Networks. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Napa, California. 2004. pp. 249–257.

[14] Corne, D. W.; Jerram, N. R.; Knowles, J. D.; et al.: PESA-II: Region-based selection in evolutionary multiobjective optimization. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*. Morgan Kaufmann Publishers Inc.. 2001. pp. 283–290.

[15] Davis, L.: Handbook of genetic algorithms. 1991.

[16] Deb, K.: *Multi-objective optimization using evolutionary algorithms*. vol. 16. John Wiley & Sons. 2001.

[17] Deb, K.; Agrawal, S.; Pratap, A.; et al.: 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.

[18] Deb, K.; Deb, K.: *Multi-objective Optimization*. Boston, MA: Springer US. 2014. ISBN 978-1-4614-6940-7. pp. 403–449.

[19] Deb, K.; Jain, H.: An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints. *IEEE Transactions on Evolutionary Computation*. vol. 18, no. 4. Aug 2014: pp. 577–601. ISSN 1089-778X.

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

[21] Defoin Platel, M.; Clergue, M.; Collard, P.: Maximum Homologous Crossover for Linear Genetic Programming. In *Genetic Programming*, *Lecture Notes in Computer Science*, vol. 2610. Springer Berlin Heidelberg. 2003. ISBN 978-3-540-00971-9. pp. 194–203.

[22] Downey, C.; Zhang, M.; Browne, W. N.: 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.

[23] Ehrgott, M.: *Multicriteria optimization*. vol. 491. Springer Science & Business Media. 2005.

[24] Estebanez, C.; Saez, Y.; Recio, G.; et al.: Automatic design of noncryptographic hash functions using genetic programming. *Computational Intelligence*. vol. 30, no. 4. 2014: pp. 798–831.

[25] Filtr, L.: Project WWW Page. `http://l7-filter.sourceforge.net/`. 2010.

[26] Floyd, R. W.; Ullman, J. D.: The Compilation of Regular Expressions into Integrated Circuits. *J. ACM*. vol. 29, no. 3. 1982: pp. 603–622.

[27] Fowler, G.; Vo, P.; Noll, L. C.: FVN Hash. Http://www.isthe.com/chongo/tech/comp/fnv/, [ONLINE, accessed: 31. 1. 2016].

[28] Gadhvi, B.; Savsani, V.; Patel, V.: Multi-objective optimization of vehicle passive suspension system using NSGA-II, SPEA2 and PESA-II. *Procedia Technology*. vol. 23. 2016: pp. 361–368.

[29] Goldman, B. W.; Punch, W. F.: Analysis of Cartesian Genetic Programming's Evolutionary Mechanisms. *IEEE Transactions on Evolutionary Computation*. vol. 19, no. 3. 2015: pp. 359–373.

[30] Grochol, D.; Sekanina, L.; Zadnik, M.; et al.: A Fast FPGA-Based Classification of Application Protocols Optimized Using Cartesian GP. In *Applications of Evolutionary Computation, 18th European Conference*. LNCS 9028. Springer International Publishing. 2015. pp. 67–78.

[31] Guo, D.; Bhuyan, L. N.; Liu, B.: An efficient parallelized L7-filter design for multicore servers. *IEEE/ACM Transactions on Networking*. vol. 20, no. 5. 2011: pp. 1426–1439.

[32] Guo, D.; Liao, G.; Bhuyan, L. N.; et al.: A scalable multithreaded l7-filter design for multi-core servers. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM. 2008. pp. 60–68.

[33] Hassan, M.; Jain, R.: *High performance TCP/IP networking*. vol. 29. Prentice Hall Upper Saddle River, NJ. 2003.

[34] Higuchi, T.; Niwa, T.; Tanaka, T.; et al.: Evolving Hardware with Genetic Learning: A First Step Towards Building a Darwin Machine. In *Proc. of the 2nd International Conference on Simulated Adaptive Behaviour*. MIT Press. 1993. pp. 417–424.

[35] Jain, H.; Deb, K.: An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point Based Nondominated Sorting Approach, Part II: Handling Constraints and Extending to an Adaptive Approach. *IEEE Transactions on Evolutionary Computation*. vol. 18, no. 4. Aug 2014: pp. 602–622. ISSN 1089-778X.

[36] Jenkins, B.: A hash function for hash Table lookup. Http://www.burtleburtle.net/bob/hash/doobs.html, [ONLINE, accessed: 31. 1. 2016].

[37] Jensen, M. T.: Reducing the run-time complexity of multiobjective EAs: The NSGA-II and other algorithms. *IEEE Transactions on Evolutionary Computation*. vol. 7, no. 5. 2003: pp. 503–515.

[38] Karagiannis, T.; Papagiannaki, K.; Faloutsos, M.: BLINC: Multilevel Traffic Classification in the Dark. *SIGCOMM Comput. Commun. Rev.*. vol. 35, no. 4. 2005: pp. 229–240.

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

[40] Kaufmann, P.; Glette, K.; Gruber, T.; et al.: Classification of Electromyographic Signals: Comparing Evolvable Hardware to Conventional Classifiers. *IEEE Tran. Evolutionary Computation,*. vol. 17, no. 1. 2013: pp. 46–63.

[41] Kaufmann, P.; Plessl, C.; Platzner, M.: EvoCaches: Application-specific Adaptation of Cache Mappings. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE Computer Society. 2009. pp. 11–18.

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

[43] Kekely, L.; Kucera, J.; Pus, V.; et al.: Software Defined Monitoring of Application Protocols. *IEEE Transactions on Computers*. vol. 65, no. 2. 2016: pp. 615–626.

[44] Kekely, L.; Pus, V.; Benacek, P.; et al.: Trade-offs and progressive adoption of FPGA acceleration in network traffic monitoring. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*. 2014. pp. 1–4.

[45] Kekely, L.; Pus, V.; Korenek, J.: Software Defined Monitoring of Application Protocols. In *Proceedings of the IEEE INFOCOM 2014 — IEEE Conference on Computer Communications*. 2014. pp. 1725–1733.

[46] Kekely, M.; Kořenek, J.: Packet Classification with Limited Memory Resources. In *In proceedings 2017 Euromicro Conference on Digital System Design*. Institute of Electrical and Electronics Engineers. 2017. ISBN 978-1-5386-2145-5. pp. 179–183.

[47] Kidoň, M.; Dobai, R.: 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.

[48] King, R. A.; Deb, K.; Rughooputh, H.: Comparison of nsga-ii and spea2 on the multiobjective environmental/economic dispatch problem. *University of Mauritius Research Journal*. vol. 16, no. 1. 2010: pp. 485–511.

[49] Knowles, J.; Corne, D.: On metrics for comparing nondominated sets. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*, vol. 1. IEEE. 2002. pp. 711–716.

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

[51] Kocsis, Z. A.; Neumann, G.; Swan, J.; et al.: Repairing and optimizing Hadoop hashCode implementations. In *International Symposium on Search Based Software Engineering*. Springer. 2014. pp. 259–264.

[52] Koza, J. R.: Genetic programming as a means for programming computers by natural selection. *Statistics and computing*. vol. 4, no. 2. 1994: pp. 87–112.

[53] Koza, J. R.: Human-competitive results produced by genetic programming. *Genetic Programming and Evolvable Machines*. vol. 11, no. 3-4. 2010: pp. 251–284.

[54] Landfeldt, B.; Sookavatana, P.; Seneviratne, A.: The case for a hybrid passive/active network monitoring scheme in the wireless Internet. In *Proceedings IEEE International Conference on Networks 2000 (ICON 2000). Networking Trends and Challenges in the New Millennium*. Sep. 2000. pp. 139–143. doi:10.1109/ICON.2000.875781.

[55] Langdon, W. B.; Harman, M.: Optimizing existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*. vol. 19, no. 1. 2015: pp. 118–135.

[56] Langdon, W. B.; Lam, B. Y. H.; Modat, M.; et al.: Genetic improvement of GPU software. *Genetic Programming and Evolvable Machines*. vol. 18, no. 1. 2017: pp. 5–44.

[57] Liao, G.; Znu, X.; Bnuyan, L.: A new server I/O architecture for high speed networks. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. 2011. ISSN 2378-203X. pp. 255–265. doi:10.1109/HPCA.2011.5749734.

[58] Lin, C.-H.; Huang, C.-T.; Jiang, C.-P.; et al.: Optimization of Pattern Matching Circuits for Regular Expression on FPGA. *IEEE Trans. Very Large Scale Integr. Syst.*. vol. 15, no. 12. 2007: pp. 1303–1310. ISSN 1063-8210.

[59] Matousek, D.; Matousek, J.; Korenek, J.: High-Speed Regular Expression Matching with Pipelined Memory-Based Automata. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. April 2018. ISSN 2576-2621. pp. 214–214.

[60] Matoušek, D.; Kubiš, J.; Matoušek, J.; et al.: Regular Expression Matching with Pipelined Delayed Input DFAs for High-speed Networks. 07 2018. pp. 104–110. doi:10.1145/3230718.3230730.

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

[62] Miller, J. F.: *Cartesian Genetic Programming*. Springer-Verlag. 2011.

[63] Miller, J. F.; Smith, S. L.: Redundancy and Computational Efficiency in Cartesian Genetic Programming. *IEEE Transactions on Evolutionary Computation*. vol. 10, no. 2. 2006: pp. 167–174.

[64] Miller, J. F.; Thomson, P.: Cartesian Genetic Programming. In *Proc. of the 3rd European Conference on Genetic Programming EuroGP2000*, *LNCS*, vol. 1802. Springer. 2000. pp. 121–132.

[65] Moore, A. W.; Zuev, D.: Internet Traffic Classification Using Bayesian Analysis Techniques. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '05. ACM. 2005. pp. 50–60.

[66] Natu, M.; Sethi, A. S.: Active Probing Approach for Fault Localization in Computer Networks. In *2006 4th IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services*. April 2006. pp. 25–33. doi:10.1109/E2EMON.2006.1651276.

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

[68] Pagh, R.; Rodler, F. F.: Cuckoo Hashing. In *Algorithms — ESA 2001*. LNCS 2161. Springer. 2001. pp. 121–133.

[69] Paxson, V.; Asanović, K.; Dharmapurikar, S.; et al.: Rethinking Hardware Support for Network Analysis and Intrusion Prevention. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Security*. HOTSEC'06. Berkeley, CA, USA: USENIX Association. 2006. pp. 11–11.
Retrieved from: http://dl.acm.org/citation.cfm?id=1268476.1268487

[70] Peterson, L. L.; Davie, B. S.: *Computer networks: a systems approach*. Elsevier. 2007.

[71] Pike, G.; Alakuijala, J.: Introducing cityhash. 2011.

[72] Press, C.: *CCNA Exploration Course Booklet: Network Fundamentals, Version 4.0*. Pearson Education India.

[73] Price, K.; Storn, R.: Differential evolution: A simple evolution strategy for fast optimization. *Dr. Dobb's journal*. vol. 22, no. 4. 1997: pp. 18–24.

[74] Rechenberg, I.: Evolution Strategy: Optimization of Technical systems by means of biological evolution. *Fromman-Holzboog, Stuttgart*. vol. 104. 1973: pp. 15–16.

[75] Rozenberg, G.; Bäck, T.; Kok, J. N.: *Handbook of natural computing*. Springer. 2012.

[76] Safdari, M.; Joshi, R.: Evolving Universal Hash Functions Using Genetic Algorithms. In *In Proc. of the Future Computer and Communication*. 2009. pp. 84–87.

[77] Schaffer, J.: Multiple Objective Optimization with Vector Evaluated Genetic Algorithms. 01 1985. pp. 93–100.

[78] Schwaller, P. J.; Bellinghausen, J. M.; Borger, D. S.; et al.: Methods, systems and computer program products for network performance testing through active endpoint pair based testing and passive application monitoring. September 23 2003. uS Patent 6,625,648.

[79] Sekanina, L.: Evolvable hardware. In *Handbook of Natural Computing*. Springer Verlag. 2012. pp. 1657–1705.

[80] Sen, S.; Spatscheck, O.; Wang, D.: Accurate, Scalable In-network Identification of P2P Traffic Using Application Signatures. In *Proceedings of the 13th International Conference on World Wide Web*. ACM. 2004. pp. 512–521.

[81] Shanthi, A. P.; Parthasarathi, R.: Practical and scalable evolution of digital circuits. *Applied Soft Computing*. vol. 9, no. 2. 2009: pp. 618–624.

[82] Sidhu, R.; Prasanna, V. K.: Fast Regular Expression Matching Using FPGAs. In *FCCM '01: Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE Computer Society. 2001. ISBN 0-7695-2667-5. pp. 227–238.

[83] Sourdis, I.; Bispo, J.; Cardoso, J. M. P.; et al.: Regular Expression Matching in Reconfigurable Hardware. *Journal of Signal Processing Systems*. vol. 51, no. 1. 2008: pp. 99–121.

[84] Srivani, L.; Giri, N. K.; Ganesh, S.; et al.: Generating synthetic benchmark circuits for accelerated life testing of field programmable gate arrays using genetic algorithm and particle swarm optimization. *Applied Soft Computing*. vol. 27. 2015: pp. 179 – 190.

[85] Stallings, W.: *High-speed networks: TCP/IP and ATM design principles*. vol. 172. Prentice hall Englewood Cliffs, NJ. 1998.

[86] Standard, S. H.: The Cryptographic Hash Algorithm Family: Revision of the Secure Hash Standard and Ongoing Competition for New Hash Algorithms. 2009.

[87] Stomeo, E.; Kalganova, T.; Lambert, C.: Generalized Disjunction Decomposition for Evolvable Hardware. *IEEE Transaction Systems, Man and Cybernetics, Part B.* vol. 36, no. 5. 2006: pp. 1024–1043.

[88] Vasicek, Z.; Bidlo, M.; Sekanina, L.: Evolution of efficient real-time non-linear image filters for FPGAs. *Soft Computing*. vol. 17, no. 11. 2013: pp. 2163–2180.

[89] Vasicek, Z.; Sekanina, L.: Formal Verification of Candidate Solutions for Post-Synthesis Evolutionary Optimization in Evolvable Hardware. *Genetic Programming and Evolvable Machines*. vol. 12, no. 3. 2011: pp. 305–327.

[90] Walker, J. A.; Trefzer, M.; Bale, S. J.; et al.: PAnDA: A Reconfigurable Architecture that Adapts to Physical Substrate Variations. *IEEE Transactiona on Computers*. vol. 62, no. 8. 2013: pp. 1584–1596.

[91] Widiger, H.; Salomon, R.; Timmermann, D.: 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.

[92] Wilson, G.; Banzhaf, W.: A comparison of cartesian genetic programming and linear genetic programming. In *Genetic Programming*. Springer. 2008. pp. 182–193.

[93] Xilinx: UltraScale Architecture and Product Overview. 2015.

[94] Xilinx Inc.: UltraScale+ FPGA, Product Tables and Product Selection Guide. Https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf, [ONLINE, accessed: 26. 7. 2019].

[95] Yoon, S.-H.; Park, J.-W.; Park, J.-S.; et al.: Internet Application Traffic Classification Using Fixed IP-Port. In *APNOMS*, *Lecture Notes in Computer Science*, vol. 5787. Springer. 2009. pp. 21–30.

[96] Yun, S.; Lee, K.: Optimization of Regular Expression Pattern Matching Circuit Using At-Most Two-Hot Encoding on FPGA. *International Conference on Field Programmable Logic and Applications*. vol. 0. 2010: pp. 40–43. ISSN 1946-1488.

[97] Zimmermann, H.: OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*. vol. 28, no. 4. April 1980: pp. 425–432. ISSN 0090-6778. doi:10.1109/TCOM.1980.1094702.

[98] Zitzler, E.; Laumanns, M.; Thiele, L.: SPEA2: Improving the strength Pareto evolutionary algorithm. *TIK-report*. vol. 103. 2001.

Related Papers

Paper I

Evolutionary Circuit Design for Fast FPGA-Based Classification of Network Application Protocols

GROCHOL David, SEKANINA Lukas, KORENEK Jan, ZADNIK Martin and KOSAR Vlastimil

In: *Applied Soft Computing*. Amsterdam: Elsevier Science, 2016, vol. 38, no. 1, pp. 933-941. ISSN 1568-4946.

Applied Soft Computing 38 (2016) 933–941

Contents lists available at ScienceDirect

Applied Soft Computing

journal homepage: www.elsevier.com/locate/asoc

Evolutionary circuit design for fast FPGA-based classification of network application protocols☆

D. Grochol, L. Sekanina*, M. Zadnik, J. Korenek, V. Kosar

Brno University of Technology, Faculty of Information Technology, IT4Innovations Centre of Excellence, Bozetechova 2, 61266 Brno, Czech Republic

ARTICLE INFO

Article history:
Received 10 July 2015
Received in revised form 1 September 2015
Accepted 24 September 2015
Available online 28 October 2015

Keywords:
Application protocol
Classifier
Cartesian genetic programming
Field programmable gate array

ABSTRACT

The evolutionary design can produce fast and efficient implementations of digital circuits. It is shown in this paper how evolved circuits, optimized for the latency and area, can increase the throughput of a manually designed classifier of application protocols. The classifier is intended for high speed networks operating at 100 Gbps. Because a very low latency is the main design constraint, the classifier is constructed as a combinational circuit in a field programmable gate array (FPGA). The classification is performed using the first packet carrying the application payload. The improvements in latency (and area) obtained by Cartesian genetic programming are validated using a professional FPGA design tool. The quality of classification is evaluated by means of real network data. All results are compared with commonly used classifiers based on regular expressions describing application protocols.

© 2015 Elsevier B.V. All rights reserved.

1. Introduction

Evolutionary algorithms (EAs) are traditionally used in the circuit design community mainly as efficient optimization techniques. In recent years, significant developments and progress in evolutionary circuit design have been witnessed. In many cases these techniques were capable of delivering efficient circuit designs in terms of an on-chip area minimization (e.g. [1]), adaptation (e.g. [2]), fabrication variability compensation (e.g. [3]), and many other properties (see, for example, many requirements on synthetic benchmark circuits in [4]). In this paper, it is exploited that the evolutionary design can produce fast and efficient circuit implementations. One of the targets is the circuit latency which is a crucial parameter in high performance computing and other applications such as security monitoring of high speed computer networks or high frequency trading. The objective of this work is to minimize the latency and area of key circuits needed in a hardware accelerator intended for classification of application protocols in high speed networks. The classifier is embedded into a software defined monitoring (SDM) platform (see details in Section 2) which is accelerated in a field programmable gate array (FPGA) [5].

In order to identify the application (or the application protocol) the network traffic belongs to, one has to inspect one or several packets with a payload. The main difficulty is that the time to process one packet is less than 7 ns in the case of modern 100 Gbps link. Hence this task has to be performed by specialized hardware. In previous work of the authors [6], key circuit components were developed for an FPGA-based application protocol classifier in which the area and latency were optimized by means of Cartesian genetic programming (CGP). The resulting circuit enabled to classify three application protocols (HTTP, SMTP, SSH) using the first packet carrying the application payload. This circuit, in fact, implemented a deterministic parallel combinational signature matching algorithm in the FPGA.

A more significant latency and area reduction, which will be crucial for classifiers supporting throughputs beyond 100 Gbps, is possible either by using advanced (faster) hardware or changing the packet processing scenario. In this paper, a new approach is proposed with respect to [6] in which small errors in the hardware protocol classification are tolerated assuming that latency and area of the classifier are significantly reduced. This concept is supported by SDM because the traffic unclassified in the hardware can be sent to the software for detailed processing.

Within this scope, the proposed work focuses on a design and optimization of three proprietary circuits, operating as application protocol classifiers, which differ in the quality of classification,

☆ This paper is an extended, improved version of the paper A Fast FPGA-Based Classification of Application Protocols Optimized Using Cartesian GP presented at EvoComNet2015 and published in: Applications of Evolutionary Computing, Proceedings of 18th European Conference, EvoApplications 2015, Copenhagen, Denmark, April 8–10, 2015, LNCS 9028, pp. 67–78, Springer, 2015.

* Corresponding author. Tel.: +420 541141215.
E-mail addresses: igrochol@fit.vutbr.cz (D. Grochol), sekanina@fit.vutbr.cz (L. Sekanina), izadnik@fit.vutbr.cz (M. Zadnik), korenek@fit.vutbr.cz (J. Korenek), ikosar@fit.vutbr.cz (V. Kosar).

http://dx.doi.org/10.1016/j.asoc.2015.09.046
1568-4946/© 2015 Elsevier B.V. All rights reserved.

latency and area. Classifier CL-acc (accuracy) is implemented according to [6] with the goal to minimize the classification error. While classifier CL-cmp (compromise) provides a moderate compromise between the latency, area and classification accuracy, classifier CL-lat (latency) is highly optimized for a low latency. Each classifier is evaluated in the task of classification of four protocols (HTTP, SMTP, SSH, and SIP) we deem most crucial from the perspective of network monitoring. It should be noted that SIP has not been considered in the initial study [6].

The main contribution of this paper is to show that these circuit classifiers can be optimized by CGP in order to significantly reduce their latency and resources requirements. The classification algorithm is not optimized by CGP. The improvements in latency (and area) obtained by CGP are validated using a professional FPGA design tool. The quality of classification is evaluated by means of real network data. All results are compared with commonly used classifiers based on regular expressions describing application protocols. Contrasted to [6], in which only key components of one classifier were implemented and optimized, complete FPGA implementations of three classifiers are evaluated.

The rest of the paper is organized as follows. Section 2 briefly surveys the field of traffic analysis in high speed networks, accelerated network technologies using FPGAs and evolutionary circuit design. Section 3 provides a specification of the classifier and network data used for the evaluation. In Section 4, the proposed hardware classifier and its approximations are introduced. Cartesian genetic programming is presented as a digital circuit design and optimization method in Section 5. Section 6 describes the implementation steps taken and the results in terms of area and latency in the FPGA. Finally, the quality of classification is assessed in terms of precision and recall. Conclusions are given in Section 7.

2. Relevant work

This paper deals with several different research areas – network traffic analysis in high speed networks, FPGA technology, fast pattern matching and evolutionary circuit design. The purpose of this section is to provide an appropriate introduction to them and to their intersections which are relevant for the target application.

2.1. Traffic analysis in high speed networks

An abstract yet detailed network traffic visibility is a key prerequisite to network management, including tasks such as traffic engineering, application performance monitoring and network security monitoring. In recent years the diversity and complexity of network applications and network threats have grown significantly. This trend has rendered monitoring of network and transport layer insufficient and it has become important to extend the visibility into the application layer, primarily to identify the application (or the application protocol) the traffic belongs to. The port numbers are no longer reliable application differentiators due to new emerging applications utilizing ports dynamicaly or to applications evading the firewalls by hiding behind well-known port numbers or utilizing port numbers defined by users [7].

The research in the area of application identification has come up with distinct approaches to identify applications carried in the traffic. These approaches differ in the level of detail that is utilized in the identification method. The most abstract one is behavioral analysis [8,9]. Its idea is to observe only the port number and destination of the connections per each host and then to deduce the application running on the host by its typical connection signature. If more details per connection are available, statistical fingerprinting [10] comes into play. In this case, a feature set is collected per each flow and the assumption is that the values of the feature set vary across applications and hence they leave a unique fingerprint. Behavioral and statistical fingerprinting generally classifies traffic to application classes rather than to particular applications. The reason is that different applications performing the same task exhibit similar behavior. For instance, application protocols such as Oscar (ICQ), MSN, XMPP (Jabber) transport interactive chat communications and hence exhibit a similar behavior, which makes it very hard to differentiate between them. The inability to distinguish applications within the same class is seen as a drawback in some situations when, for example, it is necessary to block a particular application while allowing others in the same class. The approach utilizing the greatest level of detail is a deep packet inspection. It identifies applications based on the packet payload. The payload is matched with known patterns (defined, for example, by regular expressions) derived for each application [11].

The application identification poses several on-going challenges. The identification process is bound to keep pace with ever increasing link speeds (for example, the time to process each packet is less than 7 ns in the case of a 100 Gbps link). Another challenge is represented by the growing number of protocols (i.e., the application identification must address trends such as new emerging mobile applications or applications moving into the network cloud [12]). Some deployments of application identification also require prompt (near real-time) identification to enable implementation of traffic engineering or application blocking [13].

Hardware acceleration (e.g. utilizing an FPGA) is often employed to speed up network processing [14,15], including the application identification directly on the network card. An FPGA renders it possible to utilize various pattern matching algorithms to identify applications. However, pattern matching may exhibit several constraints, that is, the high cost to process wide data inputs (which is the case for high throughput buses in FPGA) and the high complexity and overhead of a pattern matching algorithm which consumes valuable hardware resources or constrains the achievable frequency.

These drawbacks are addressed by alternative methods which look for constants and fixed-length strings (for brevity they are called the *signatures* in the paper) rather than regular expressions (e.g. [16]). This paper builds upon this strategy and envisions a hardware-software codesign approach in which a simple circuit labels the traffic belonging to applications of interest with some probability of false positives while software can subsequently handle and check the labeled traffic with a more complex algorithm effectively. This approach is supported by the software defined monitoring concept [5]. Software defined monitoring employs sophisticated processes running in the software to subsequently install rules in the hardware (network card). While it is not possible (or at a very high cost) to process all traffic in the software, the application identification is offloaded into the hardware. The offload not only reduces the host memory and processor load but it also increases the expressive strength of the SDM rules. The target applications range from application-specific forwarding and traffic shaping to traffic monitoring and blocking.

2.2. FPGAs in network applications

Performance requirements are growing due to the increasing volume and rates of network traffic. Paxson et al. [17] argue that these performance requirements should be met by leveraging a high degree of possible parallelism that is inherent to network traffic monitoring. FPGAs as well as ASICs may deliver such a vast support of parallelism. However, only FPGAs render it possible to prototype and implement critical application components for various network applications at the highest speeds while the optimized ASICs follow after broad deployment a few years later on. FPGAs are thus extensively used in the so-called hardware-accelerated

network cards to implement the first line of network traffic processing, such as monitoring, forwarding and other applications [18,19].

FPGAs include a high spectrum of components, but the following components are crucial for the purposes of this paper. FPGAs consist of routing network and basic building blocks such as look-up tables (LUTs), registers and block memories. The particular setup of the routing network defines the interconnection of these components (i.e. the layout of the circuit). The LUTs serve to implement combinational logic while registers and block memories serve to keep the stateful information. Modern FPGAs contain millions of LUTs and registers and thousands of block memories. All these components may, in theory, work in parallel independet of each other providing enormous computation power with a low energy consumption in tens of Watts. Moreover, FPGAs targeting the network market include more than a hundred of high-speed transceivers allowing for connection to high speed network links (e.g. high-end Virtex UltraScale+ FPGA offers up to 4Tbps of aggregated transceiver throughput [20]). The crucial task is to transform a high-level description of the circuit (for example, written in VHDL or SystemC) into an effective implementation in FPGA from the perspective of meeting the timing and resource constraints.

2.3. Fast pattern matching

The L7 filter [21] is a popular program for application protocol identification, which utilizes regular expressions to describe application protocols. It performs pattern matching in network flows. If a known pattern is matched in the payload, the corresponding application protocol is assigned to the network flow. Current processors are not powerful enough to achieve 100 Gbps throughput for regular expression matching. The throughput of L7 decoder is less than 1 Gbps per one CPU core even for the latest Xeon processors. In order to achieve 100 Gbps throughput, it is necessary to use highly optimized hardware architectures.

In recent years, many researchers have proposed high-speed pattern matching hardware architectures, which utilize the fine grained parallelism of FPGA technology. Mapping of regular expressions matching to an FPGA was first explored by Floyd and Ullman [22], who showed that a Nondeterministic Finite Automaton (NFA) can be implemented using a programmable logic array. Sindhu et al. [23] proposed efficient mapping of NFAs to FPGA and Clark et al. improved the mapping by a shared decoder [24,25] which significantly reduced the amount of consumed logic resources. The AMTH (At Most Two-Hot encoding) [26] architecture improves NFA mapping to the FPGA. The combination of one-hot and binary encoding reduces the amount flip-flops, which represent NFA states.

Several papers introduced optimized mapping of Perl Compatible Regular Expressions (PCRE), which are widely used in Intrusion Detection Systems (IDS). Sourdis et al. published in [27] an architecture that allows for the sharing of character classes, static subpatterns and introduced components for efficient mapping of constrained repetitions to the FPGA. Lin et al. created an architecture for sharing infixes and suffixes [28]. Nevertheless, these optimizations are relevant only for large sets of PCRE in IDS systems. In this work, a small set of regular expressions without counting constraints and other advance PCRE constructions is only used. Therefore, these optimizations are not considered in the evaluation of proposed architectures.

The throughput of pattern matching is determined by the amount of bytes processed within one clock cycle and frequency of the hardware matching unit. The FPGA technology limits the maximum frequency to several hundreds of MHz. To increase the processing speed, the NFA can be modified to process multiple bytes per one clock cycle [29]. Unfortunately, with the increasing size of the NFA input, the amount of NFA transitions grows exponentially.

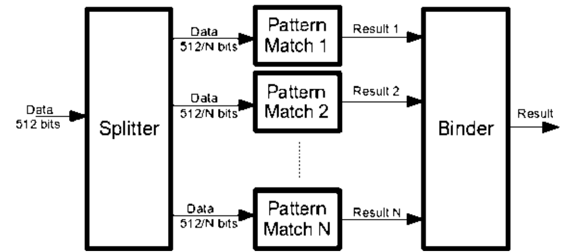


Fig. 1. Increasing the throughput by multiple pattern matching units.

As a result, the hardware matching unit consumes much more FPGA resources and the frequency decreases rapidly.

The throughput can be increased by multiple parallel matching units. These units need additional logic resources and buffers to distribute network data to the matching units and join the results. The overhead of parallel processing is shown in Fig. 1. First, the splitter has to assign the sequence number into every packet and store the packet to the buffer. The packet data are then sent with a lower rate to parallel matching units. The units perform pattern matching and provide the results to the binder, which needs buffers to order the results in the right sequence order.

It can be seen that the parallel matching units can scale the matching speed up to 100 Gbps throughput, but only at the cost of significant overhead in terms of latency, FPGA logic resources and memory buffers. This overhead is avoided by focusing on highly optimised hardware architectures with high throughput and low latency.

2.4. Evolutionary circuit design

The idea of evolvable hardware and automated circuit design by means of artificial evolution was introduced by Higuchi et al. in 1993 [30]. A recent survey of the field covering key subfields (evolutionary hardware design and adaptive hardware) is available in [31]. Significant progress in the evolution of digital circuits is connected with Cartesian genetic programming which has been developed by Miller since 1999 and utilized in many applications as documented in the recent monograph [32]. Since only combinational circuits will be evolved in this work, CGP is a natural choice.

CGP is a form of genetic programming in which candidate designs are represented using directed oriented graphs (see a detailed description in Section 5). In the standard CGP used for combinational circuit evolution, each candidate circuit is directly mapped into a chromosome consisting of a string of integers and evaluated by applying all possible input vectors. Although various new designs have been discovered using CGP, the method is not directly applicable for the design of large combinational circuits because the fitness evaluation time grows exponentially with the number of primary inputs. Moreover, the number of evaluations can easily go into the millions, even for small (but non-trivial) circuits such as multipliers. This problem has partially been eliminated by introducing circuit decomposition techniques at the representation level [33,34] and formal verification methods in the fitness function [1]. Other successful applications of CGP have been proposed in domains in which candidate circuits are not evaluated using all possible input combinations (see e.g. hash functions [35], image operators [36] or classifiers [2]).

In order to accelerate the fitness function evaluation on a common processor, a bit-level parallel simulation of candidate combinational circuits is employed. Contrasted to a naïve simulation, in which 2^k vectors are sequentially submitted for evaluation (where k is the number of primary inputs), the bit-level parallel simulation exploits the fact that current processors enable performing bitwise operations over two w-bit operands in parallel.

Table 1
The flows corresponding to the application protocols in data sets.

Data set	CESACO		CESPIO		DATASET SIP	
Protocol	Count flows	Count flows [%]	Count flows	Count flows [%]	Count flows	Count flows [%]
HTTP	1914	38.12	15060	52.29	134	2.41
SMTP	4	0.08	34	0.12	10	0.18
SSH	1	0.02	0	0.00	14	0.25
SIP	0	0	0	0	5204	93.42
Others	3102	61.78	13705	47.59	208	3.74
All	5021	100.00	28799	100.00	5570	100.00

Hence the input vectors are grouped into w-bit words and simulated in parallel. The obtained speedup is w on a w-bit processor, for example, 64 on a common personal computer. Even if this approach is taken, a typical CGP run could take tens of minutes for a circuit with 8 inputs and 8 outputs.

There are only a few papers dealing with evolutionary circuit design at the level of 4-input LUTs [35,37] and no paper dealing with 6-input LUTs. Unfortunately, the bit-level parallel simulation is inefficient for circuits consisting of LUTs because their logic function has to be emulated using a sequence of binary logic operations. Moreover, employing CGP with 6-input LUTs (each of them encoded using 64 bits in the chromosome) would lead to long chromosomes, complex search spaces and very inefficient search procedures. Hence two-input gates represent the dominant option when CGP is applied to the evolution of complex circuits.

3. Requirements and network data

In order to design, implement and evaluate an FPGA-based application protocol classifier, its basic parameters and an environment in which it will be operated have to be specified.

3.1. Specification of the classifier

The classifier has to distinguish among four application protocols (HTTP, SMTP, SSH and SIP) which represent an important portion of the network traffic and play an important role in traffic monitoring. Remaining protocols will be classified as unknown. Because the primary goal is achieving a very low latency, only signatures of the first packet carrying the application payload will be defined and utilized in the classifier architecture. The classifier will operate in an FPGA on a 512 bit bus to meet the 100 Gbps throughput. The application payload may start at nearly arbitrary offset (byte of a word) on the bus and the application (protocol) must be identified each clock cycle to keep pace even with the shortest incoming packets of 64 bytes.

The classifier will be constructed manually – as a combinational circuit with a low latency. CGP will be applied to optimize its key subcircuits to reduce the latency and area. An observation is utilized that a circuit which is well optimized by a commercial FPGA synthesis tool can further be re-synthesized and re-optimized by CGP to improve its parameters (see example circuits created by this approach in [36]). Such a classifier will be considered as a fully functional solution (CL-acc).

Further area and latency improvement are obtained if the requirement of full functionality can be relaxed. Hence we will also propose and evaluate classifiers (CL-cmp and CL-lat) showing a shorter latency and smaller area. Providing such approximations is currently a hot topic in computer engineering. The approach is called *approximate computing* and its goal is to investigate how computer systems can be made better – more energy efficient, faster, and less complex by relaxing the requirement that they are exactly correct [38].

3.2. Network data

The data which has to be classified are common network data (available in the pcap format). In our case, complete network data sets with anonymized IP addresses are utilized, collected on CESACO link (connecting CESNET and ACONET networks) and CESPIO link (connecting CESNET and PIONIER networks), see Table 1. Because SIP and SSH are not adequately present in these data sets, another, dedicated data set (DATASET SIP) with a high presence of SIP records was employed.

For example, the available record from CESPIO contains 43 M packets, where percentages are 78.72% for TCP, 20.58% for UDP, 0.18% for ICMP and 0.53% others. One can observe that only TCP and UDP are relevant for our purposes. The packet traces were analyzed using Scapy. In the case of HTTP, SMTP and SSH, which operate over TCP, the third or the fourth packet of the TCP connection is usually considered as the first packet containing the application payload. The L7 filter [21] was utilized as a reference classifier to annotate each connection in the data set.

The resulting data sets, which can be used for evaluation purposes, are available in the JSON format. Each record contains the source IP and port, the destination IP and port, the transport protocol number, and the whole packet encoded using base64 (see Fig. 2). Table 1 gives the mix of considered protocols in our data sets.

4. Proposed classifiers

This section describes the analytical approach taken in order to construct the proposed classifiers. Detailed hardware architecture of the classifiers is then presented.

4.1. Deterministic classification

Because the classification utilizes only the start of the payload, several initial bytes of considered application protocols were analyzed and characters were identified which are unique in these protocols. Table 2 shows the unique *signatures* that were identified for considered protocols. The longest signature of the CL-acc contains 10 characters (bytes). Signatures of classifier CL-cmp are constructed from those used in CL-acc in such a way that they are reduced to the first 4 characters, which leads to less complex hardware. Further area and latency reduction is expected in classifier CL-lat which operates with signatures containing at least 3 characters, but each of them has to exist in at least two signatures of CL-acc.


```
{
"dIP": "192.168.0.2",
"dPort": 80,
"data": "R0VUIC9zaXRlcy9kZWZhdWx0L3RoZW1lcy9mcmFtZWR5bmFtaWMv...
"id": "(' 192.168.0.1', '192.168.0.2', 52217, 80)",
"trProto": 6,
"protocol": "HTTP",
"sIP": " 192.168.0.1",
"sPort": 52217
},
```


Fig. 2. Example of record in the data set.

Table 2
Unique signatures in considered application protocols.

Protocol	CL-acc	CL-cmp	CL-lat
HTTP	"GET /"	"GET "	"*ET /"
	"PUT /"	"PUT "	"*UT /"
	"POST /"	"POST"	"*OS* /"
	"HEAD /"	"HEAD"	"*EA* /"
	"TRACE /"	"TRAC"	"T*ACE**"
	"DELETE /"	"DELE"	"*E**TE**"
	"OPTIONS /"	"OPTI"	"**TI*NS**"
SIP	"INVITE "	"INVI"	"*N*ITE"
	"REGISTER "	"REGI"	"*E*IS*E*"
	"CANCEL "	"CANC"	"C**CE*"
	"MESSAGE "	"MESS"	"*ESS**E"
	"SUBSCRIBE "	"SUBS"	"SU*SC***E*"
	"NOTIFY "	"NOTI"	"*OTI**"
SSH	"SSH-"	"SSH-"	"SS*-"
SMTP	"220 "	"220 "	"220 "
	"220-"	"220-"	"220-"

These classifiers can be constructed as combinational circuits by means of a decoder. However, they have to correctly manage the cases in which the signatures appear at various offsets within the frame due to preceding protocol headers, which is a natural situation in real network traffic data.

4.2. Classifiers in hardware

The hardware architecture utilizes a 512 bit bus to transfer protocol frames. Each frame starts with the headers of low-level protocols such as Ethernet, IPv4 or IPv6, TCP or UDP. As a result, the start of the application payload may appear with certain offsets on the bus, namely 2 bytes from the position 0 or with $2+4k$ bytes, where $k=1, \ldots, 16$.

All three versions of the classifier are constructed according to Fig. 3 which also shows that the circuit classifier consists of three levels of combinational logic.

Table 3
CL-acc: mapping functions in the coders. The * symbol means: "not utilized in a particular coder". ω stands for "otherwise".

Coder 1	Coder 2	Coder 3	Coder 4	Output
Space	Space	Space	Space	00000011
/	/	/	/	00000101
2	2	0	–	00000110
A	A	A	B	00001001
C	E	B	C	00001010
D	G	E	D	00001100
E	L	G	E	00010001
F	N	H	I	00010010
G	O	I	R	00010100
H	P	L	S	00011000
I	R	N	T	00100001
M	S	S	*	00100010
N	T	T	*	00100100
O	U	V	*	00101000
P	Y	*	*	00110000
R	*	*	*	01000001
S	*	*	*	01000010
T	*	*	*	01000100
ω	ω	ω	ω	00000000

In the first level, one coder is connected to each byte of the word (64 coders, in total). There are four types of the coders (c1, c2, c3, c4) because of the 4-byte offsets. Each coder implements a mapping from the set of characters allowed for the given position to a set of 8-bit values in which just 2 bits are not zeros. The mapping functions of the coders in CL-acc, CL-cmp and CL-lat are given in Tables 3–5.

This remapping implemented by coders allows for a fast signature detection in the subsequent level of comparators. All possible occurrences of the application data within the input word are thus processed in parallel.

The second level consists of comparators. In the case of CL-acc, each of them compares the outputs of ten coders (note that the longest signature contains 10 characters) with the unique patterns identified for the considered application protocols. If a particular application protocol is detected then its 4-bit code is visible at the output of the comparators (0001 – HTTP, 0010 – SMTP, 0100 – SSH,

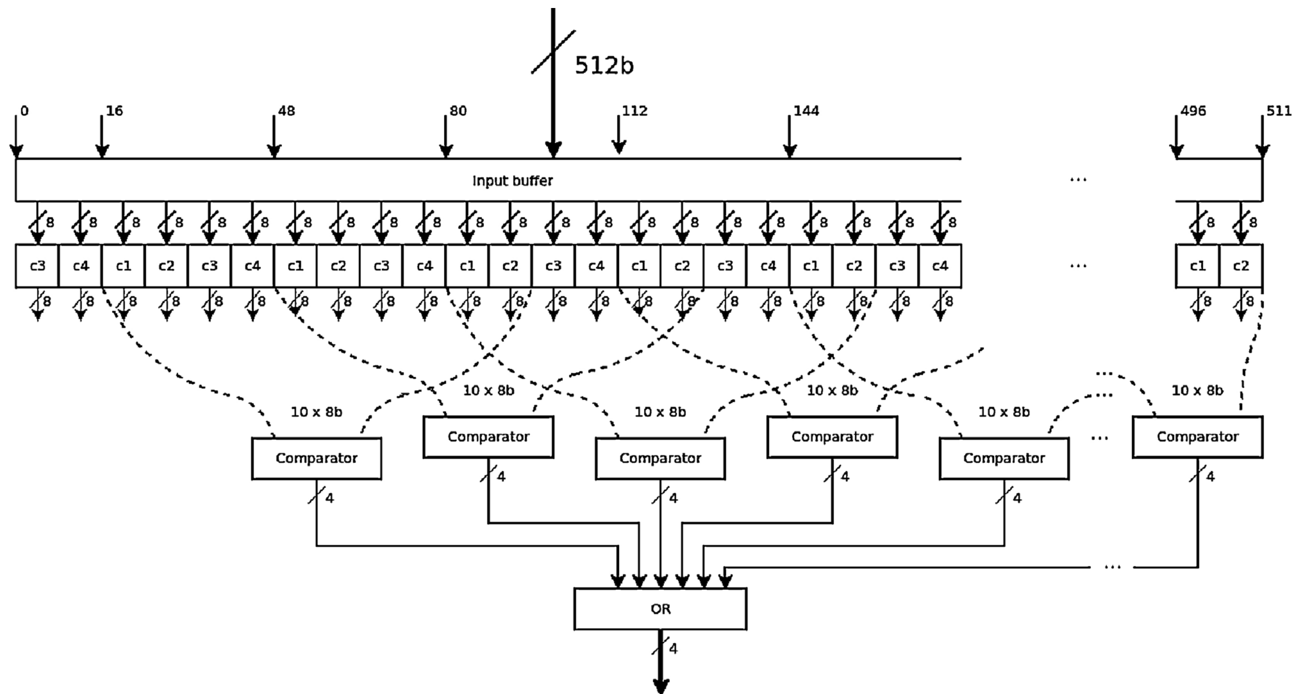


Fig. 3. Classifier CL-acc as a combinational circuit.

Table 4
CL-cmp: mapping functions in the coders.

Coder 1	Coder 2	Coder 3	Coder 4	Output
2	2	0	Space	00000011
C	A	A	–	00000101
D	E	B	C	00000110
G	N	G	D	00001001
H	O	H	E	00001010
I	P	L	I	00001100
M	R	N	S	00010001
N	S	S	T	00010010
O	U	T	*	00010100
P	*	V	*	00011000
R	*	*	*	00100001
S	*	*	*	00100010
T	*	*	*	00100100
ω	ω	ω	ω	00000000

Table 5
CL-lat: mapping functions in the coders.

Coder 1	Coder 2	Coder 3	Coder 4	Output
Space	/	Space	Space	00000011
/	2	0	–	00000101
2	E	A	C	00000110
C	N	E	I	00001001
E	S	S	S	00001010
S	O	T	*	00001100
T	U	*	*	00010001
ω	ω	ω	ω	00000000

1000 – SIP, 0000 – unknown). In the case of CL-cmp (CL-lat, respectively) the circuit is simplified as only 4 (9, respectively) coders are employed. Finally, at the third level, all 4-bit codes are fed to an OR gate which indicates a presence of the detected application protocols or unknown protocol (0000).

5. Coder evolution using CGP

Based on our previous experience, it is assumed that parameters of a circuit optimized by a professional FPGA design software can be improved if CGP is employed [36]. As the whole classifier is a relatively complex circuit to be optimized, it is proposed to evolve its components – 64 (combinational) coders. Each of the coder types c1, c2, c3 and c4 will be evolved by CGP separately. The standard CGP is used as defined in [32].

In CGP, a candidate circuit is modeled as a directed acyclic graph and represented in a 2D array of $n_c \times n_r$ processing nodes. Each node is capable of performing one of the n_a-input functions specified in Γ set. The setting of n_c, n_r and Γ significantly influences the performance of CGP [39,40]. Current FPGAs utilize 6-input LUTs as building blocks of all circuits. However, employing CGP with 6-input nodes (each of them encoded using $2^6 = 64$ bits in the chromosome) would lead to long chromosomes, complex search spaces and so inefficient search procedures. It is proposed to optimize the coders at the level of 2-input nodes (encoded using up to 4 bits) and let the professional circuit synthesis software implement the resulting optimized circuits using 6-input LUTs in the FPGA.

The remaining parameters of CGP are the number of primary inputs (n_i), the number of primary outputs (n_o), and the level-back parameter (L) specifying which nodes can be used as inputs for a given gate. The primary inputs and the outputs of nodes are labeled $0 \ldots n_c \cdot n_r + n_i - 1$ and considered as addresses which connections can be fed to. In the chromosome, each two-input node is then encoded using three integers (an address for the first input; an address for the second input; a node function). Finally, for each primary output, the chromosome contains one integer specifying the connection address. Fig. 4 shows an example and a corresponding chromosome.

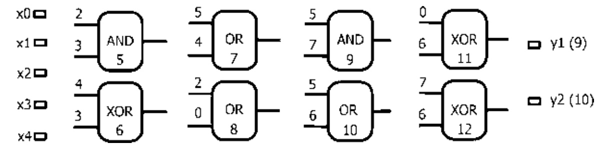


Fig. 4. Example of a combinational circuit in CGP with parameters: $n_i = 5$, $n_o = 2$, $L = 4$, $n_c = 4$, $n_r = 2$, $\Gamma = \{$AND (0), OR (1), XOR (2)$\}$. Gates 8, 11 and 12 are not utilized. Chromosome: 2,3,0; 4,3,2; 5,4,1; 2,0,1; 5,7,0; 5,6,1; 0,6,2; 7,6,2; 9, 10. The last two integers indicate the outputs of the circuit.

The chromosome size is $(n_a + 1)n_r n_c + n_o$ genes (integers). The main feature of this encoding is that the size of the chromosome is constant for a given n_i, n_o, n_a, n_r and n_c. However, the size of circuits represented by such chromosomes is variable as some nodes can remain disconnected. The nodes which are included into the circuit after reading the chromosome are called the active nodes.

The search is performed using a simple search strategy $(1 + \lambda)$, where λ is the number of offspring circuits created by mutation from one parent [32]. The initial population is randomly generated. A new population consisting of λ individuals is generated by applying the mutation operator on the best individual of the previous population. The mutation operator randomly modifies h integers of the chromosome. The evolution is terminated after producing a given number of generations.

In the case of combinational circuits, the fitness value of a candidate circuit is defined as [31]

$$f = \begin{cases} b & \textit{when} \quad b < n_o 2^{n_i}, \\ b + (n_c n_r - z) & \textit{otherwise}, \end{cases} \tag{1}$$

where b is the number of correct output bits obtained as response for all possible assignments to the inputs, z denotes the number of gates utilized in a particular candidate circuit and $n_c n_r$ is the total number of available gates. It can be seen that the last term $n_c n_r - z$ is considered only if the circuit behavior is perfect, i.e. $b = b_{max} = n_o 2^{n_i}$. The second term can be modified to optimize other circuit parameters.

Latency is one of the key parameters of classification. After performing numerous experiments which are reported in Section 6.2 as well as in [6], it was recognized that the minimum latency is 12Δ (where Δ is delay of a two-input gate) if fully functional coders are requested. Hence latency is not explicitly optimized in our approach; however, its maximum value is implicitly determined by $n_c = 12$.

6. Results

The experimental evaluation consists of the following steps: (1) conventional implementation of the proposed classifiers; (2) CGP-based optimization of selected subcomponents (coders); (3) resynthesis of the classifiers with optimized subcomponents; (4) verification of the quality of classification.

6.1. Conventional implementation

Three circuits corresponding to classifiers CL-acc, CL-cmp and CL-lat were behaviorally described in VHDL and synthesized into the Xilinx Virtex-7 XC7VH580T FPGA using Xilinx ISE Project navigator 14.4 tool. The target FPGA contains 6-input LUTs whose latency is 0.043 ns. The circuit latency was set as the main optimization target for the synthesis tool. Parameters of the resulting circuits which are considered as reference conventional implementations in the context of this paper are given in Table 6. One can observe

Table 6
Results of synthesis for the Xilinx Virtex-7 XC7VH580T FPGA.

Classifier	LUTs	Flip flop	Latency [ns]
CL-acc	2352	0	6.410
CL-acc + CGP	1909	0	6.113
CL-cmp	1549	0	6.093
CL-cmp + CGP	1073	0	5.604
CL-lat	1625	0	5.943
CL-lat + CGP	1217	0	5.139
Yamagaki/Clark	10,431	2326	77.504 (16 × 4.844)
AMTH	10,547	2190	71.536 (16 × 4.671)

that CL-lat is less complex and faster than CL-cmp and CL-cmp is less complex and faster than CL-acc.

6.2. Optimization by CGP

There are four types of coders in each of the three classifier circuits (Fig. 3). These 8-input/8-output coders are optimized by CGP operating at the gate level. The setting of CGP parameters is forced by the specification or it can be considered as typical for CGP. The reasons for the chosen parameter values are as follows: $n_i = 8$ and $n_o = 8$ directly follows from the specification; $n_c = 12$ reflects our strategy to restrict the maximum delay to 12Δ; $n_r = 50$ is used to provide a sufficient redundancy at the level of genotype assuming that evolved circuits will contain about 30–50 active gates [39]; L is restricted to one logic level in order to generate compact circuits and enable the deep pipeline processing in future implementations; $\lambda = 4$ is a recommended value for CGP [39,32]; and $h = 5$ corresponds with the mutation probability $5/(12 \cdot 50) = 0.00833$, i.e. with the typical values $0.001 - 0.01$ used for the mutation across almost all other studies [32]. In the case of determining the function set, we compared CGP utilizing all logic functions over two inputs except logic constants (which will be denoted Γ) against a reduced function set containing logic functions $\{a, b, \neg a, \neg b, a \vee b, a \wedge b, a \oplus b\}$. In addition to the completeness of the reduced function set (i.e. $\neg$, $\vee$ and $\wedge$ are included), the xor function ($\oplus$) is supported to enable the xor decomposition which is very useful for optimizing the xor intensive logic functions. The initial population is seeded using randomly generated circuits.

In total, circuits for 24 specifications (3 classifiers × 4 coders × 2 versions of function set) were evolved. In order to obtain basic statistics, each run consisting of 5 million generations was repeated 20 times. The number of gates, obtained at the end of CGP runs devoted to a particular specification, are presented in form of boxplots in Fig. 5 (CL-acc), Fig. 6 (CL-cmp) and Fig. 7 (CL-lat). Boxplots used in these figures contain the minimum, first quartile, median, third quartile and maximum.

The experiments confirmed our assumption that the optimized coders of CL-lat are less complex than those optimized for CL-cmp and CL-acc. It can also be seen that the usage of the complete function set consistently gives more compact coders than the reduced function set despite the fact that the search space is more complex.

In order to determine the impact of the CGP optimization to subsequent circuit synthesis and optimization conducted by means of a professional FPGA design tool, VHDL implementations of all coders evolved by CGP for CL-acc were developed and synthetized for the FPGA. The number of LUTs is presented in form of boxplots in Fig. 8. The most important observation is that the most compact FPGA implementations of coders are obtained if the circuit description entering the FPGA synthesis process contains the gates from reduced function set. It is quite unintuitive with respect to boxplots shown in Fig. 5–7. This interesting result deserves further investigations which are beyond the scope of this paper.

An analysis of optimized circuits is presented for c2 which is a middle-size coder. When optimized for the accurate CL-acc, the

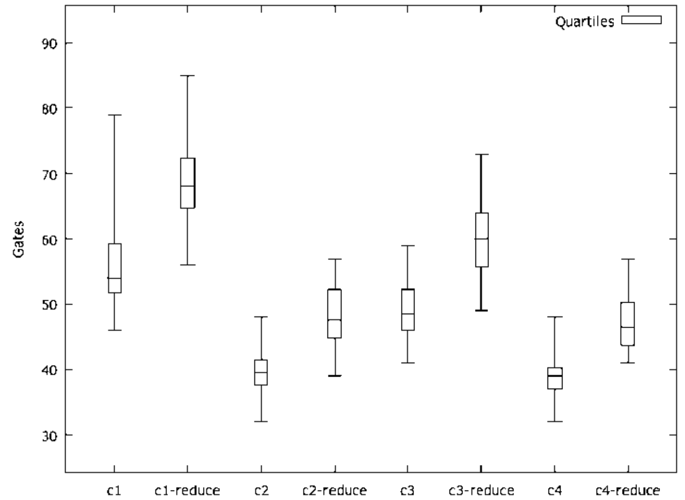


Fig. 6. The number of gates obtained at the end of 20 CGP runs for four coders of classifier CL-cmp. 'Reduce' stands for 'reduced set of gates'.

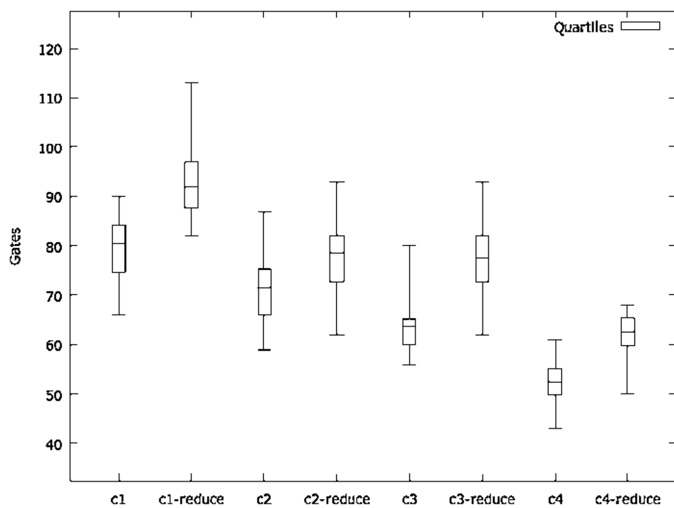


Fig. 5. The number of gates obtained at the end of 20 CGP runs for four coders (c1, c2, c3 and c4) of classifier CL-acc. 'Reduce' stands for 'reduced set of gates'.

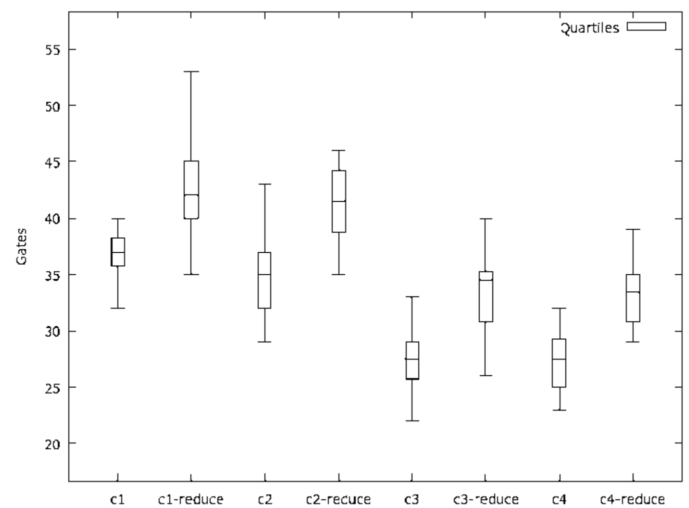


Fig. 7. The number of gates obtained at the end of 20 CGP runs for four coders of classifier CL-lat. 'Reduce' stands for 'reduced set of gates'.

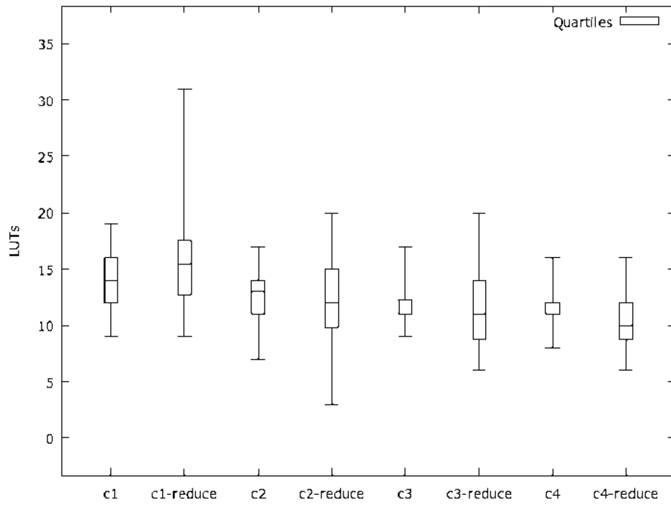


Fig. 8. The number of LUTs obtained for four coders of classifier CL-acc. 'Reduce' stands for 'reduced set of gates'.

most compact implementation of c2 consists of 59 gates (62 gates in the case of the reduced set of gates). When it is approximated for CL-lat, the most compact implementation requires only 29 gates (35 gates in the case of the reduced set of gates), which is an important reduction. At the level of LUTs, the reduction provided for CL-lat is not so remarkable because only one LUT was saved (7 versus 6 LUTs) if the FPGA synthesis starts with circuits evolved using the full gate set (Γ). However, if the FPGA synthesis starts with circuits evolved for the reduced gate set, the resulting implementation contains 3 LUTs (for CL-acc) and 2 LUTs for the most relaxed case (CL-lat). Considering the fact that 64 coders have to be implemented into the FPGA, the obtained resources reduction is significant.

6.3. Classifier resynthesis using optimized coders

The most compact implementations of coders were translated to VHDL and utilized in the VHDL code of classifiers CL-acc, CL-cmp and CL-lat. These modified classifiers were synthesized with the same setting as reported in Section 6.1.

The results of synthesis are labeled using '+CGP" and given in Table 6. Both crucial circuit parameters (latency and area expressed as the number of LUTs) were significantly improved by CGP for all classifiers.

Enabling approximate classification (CL-lat and CL-cmp) whose implementation is further optimized by CGP led to 48.2% improvement in area (LUTs) and 19.8% improvement in latency with respect to a solution (CL-acc) which would be produced by a conventional signature-based approach.

In order to compare the proposed solution with the state of the art classifiers from the literature, parameters of Yamagaki/Clark and AMTH circuit classifiers were included to Table 6. These classifiers accurately implement the L7-filter (for considered protocols) by means of optimized finite state machines. The main conclusion is that CL-lat optimized by CGP exhibits the area (LUTs) and latency one order of magnitude lower than Yamagaki/Clark and AMTH.

6.4. Quality of classification

The quality of classification was evaluated offline, utilizing a software model that has been developed for the proposed classifiers. The evaluation was performed using all three data sets in which we considered traces containing first payload packets. The output of our classifiers was verified against the L7 filter which provides 100% correct results for considered protocols. Precision and Recall metrics were calculated:

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} \tag{2}$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} \tag{3}$$

Precision informs us how many packets assigned to a given class are truly correctly assigned. Fig. 9 shows the quality of classification for the worst case – classifier CL-lat. It can be seen that HTTP, whose representation is rich in our data sets, is classified perfectly. The reason for lower percentages of Precision in the case of SMTP is the

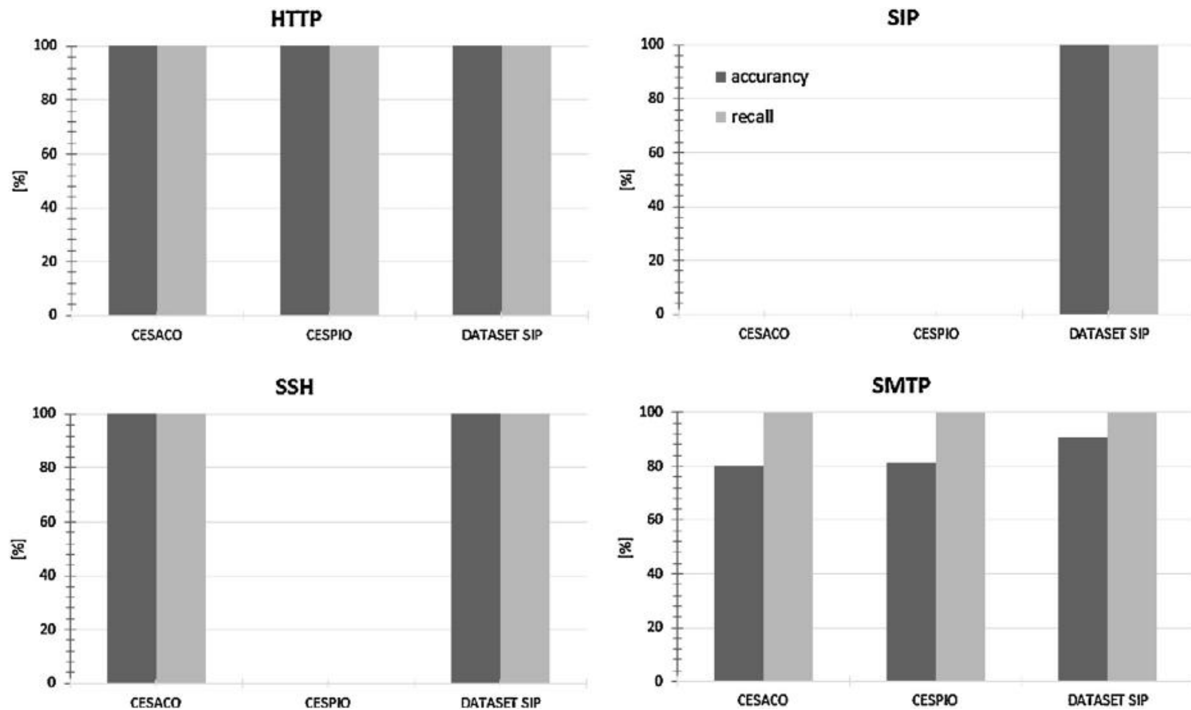


Fig. 9. Precision and Recall percentages for four classified protocols on three data sets obtained using CL-lat.

fact that considered signatures are relatively short and can easily appear inside of other protocol packets. As the subsequent packet processing is done in software precisely the incorrectly classified protocols will be recognized anyway. The software task is simpler than that of the original one. The software must only verify the labelled traffic and dismiss false positives.

Considering the whole SDM, which the proposed classifiers are targeted for, the Recall is even a more important metrics. High Recall values indicate that if a given application protocol is present in the traffic data, it is detected with almost 100% probability and thus no information is lost. Fig. 9 does not give any data for SSH in CESACO and SIP in CESACO and CESPIO. The reason is that there are no relevant records in these data sets.

7. Conclusions

It was shown how evolved circuits, optimized for the latency and area, can significantly increase the throughput of a manually designed classifier of application protocols. This paper introduced a new concept of hardware classifier which is composed as a fast combinational circuit performing signature matching where the signatures are designed according to the protocols to be classified. Its accurate implementation (CL-acc) was then relaxed and approximate classifiers CL-cmp and CL-lat were proposed with reduced area and latency. Key components of all classifiers were optimized by CGP with the aim of further area and latency reduction. This led to 48.2% improvement in area (LUTs) and 19.8% improvement in latency with respect to CL-acc. Finally, the proposed classifiers were compared with state of the art circuits accurately implementing the L7-filter and reported improvement in area and latency by one order of magnitude. The proposed solution is capable of a fast detection of key application protocols using a single packet only. It exhibits excellent Recall values (no monitored application protocols are missed). The proposed classifier will be used in the SDM framework, which will handle detailed packet processing to improve the precision parameter of the hardware classifier.

Acknowledgments

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

References

[1] Z. Vasicek, L. Sekanina, Formal verification of candidate solutions for post-synthesis evolutionary optimization in evolvable hardware, Genet. Program. Evol. Mach. 12 (3) (2011) 305–327.
[2] P. Kaufmann, K. Glette, T. Gruber, M. Platzner, J. Torresen, B. Sick, Classification of electromyographic signals: comparing evolvable hardware to conventional classifiers, IEEE Tran. Evol. Comput. 17 (1) (2013) 46–63.
[3] J.A. Walker, M. Trefzer, S.J. Bale, A.M. Tyrrell, Panda: a reconfigurable architecture that adapts to physical substrate variations, IEEE Trans. Comput. 62 (8) (2013) 1584–1596.
[4] L. Srivani, N.K. Giri, S. Ganesh, V. Kamakoti, Generating synthetic benchmark circuits for accelerated life testing of field programmable gate arrays using genetic algorithm and particle swarm optimization, Appl. Soft Comput. 27 (2015) 179–190.
[5] L. Kekely, J. Kucera, V. Pus, J. Korenek, A. Vasilakos, Software defined monitoring of application protocols, IEEE Trans. Comput. (2015) 1–14, http://dx.doi.org/10.1109/TC.2015.2423668.
[6] D. Grochol, L. Sekanina, M. Zadnik, J. Korenek, A fast FPGA-based classification of application protocols optimized using Cartesian GP, in: Applications of Evolutionary Computation, 18th European Conference, LNCS 9028, Springer International Publishing, 2015, pp. 67–78.
[7] T. Karagiannis, A. Broido, N. Brownlee, K. Claffy, M. Faloutsos, Is P2P dying or just hiding? in: Global Internet and Next Generation Networks, Globecom 2004, Dallas, Texas, 2004.
[8] T. Karagiannis, K. Papagiannaki, M. Faloutsos, Blinc: multilevel traffic classification in the dark SIGCOMM Comput. Commun. Rev. 35 (4) (2005) 229–240.
[9] S.-H. Yoon, J.-W. Park, J.-S. Park, Y.-S. Oh, M.-S. Kim, Internet application traffic classification using fixed IP-port, in: APNOMS, Vol. 5787 of Lecture Notes in Computer Science, Springer, 2009, pp. 21–30.
[10] A.W. Moore, D. Zuev, Internet traffic classification using Bayesian analysis techniques, in: Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '05, ACM, 2005, pp. 50–60.
[11] S. Sen, O. Spatscheck, D. Wang, Accurate, scalable in-network identification of p2p traffic using application signatures, in: Proceedings of the 13th International Conference on World Wide Web, ACM, 2004, pp. 512–521.
[12] A. Tongaonkar, R. Keralapura, A. Nucci, Challenges in network application identification, in: Presented as Part of the 5th USENIX Workshop on Large-Scale Exploits and Emergent Threats, USENIX, Berkeley, CA, 2012.
[13] L. Bernaille, R. Teixeira, K. Salamatian, Early application identification, in: Proceedings of the 2006 ACM CoNEXT Conference, ACM, New York, NY, USA, 2006, pp. 6:1–6:12.
[14] N. Zilberman, Y. Audzevich, G. Covington, A. Moore, NetFPGA SUME: toward 100 Gbps as research commodity, Micro, IEEE 34 (5) (2014) 32–41.
[15] S. Friedl, V. Pus, J. Matousek, M. Spinler, Designing a Card for 100 Gb/s Network Monitoring, Tech. Rep., CESNET, 2013.
[16] B.-C. Park, Y. Won, M.-S. Kim, J. Hong, Towards automated application signature generation for traffic identification, in: Network Operations and Management Symposium, 2008. NOMS 2008, IEEE, 2008.
[17] V. Paxson, K. Asanović, S. Dharmapurikar, J. Lockwood, R. Pang, R. Sommer, N. Weaver, Rethinking hardware support for network analysis and intrusion prevention, in: Proceedings of the 1st USENIX Workshop on Hot Topics in Security, HOTSEC'06, USENIX Association, Berkeley, CA, USA, 2006, p. 11 http://dl.acm.org/citation.cfm?id=1268476.1268487.
[18] G. Antichi, S. Giordano, D. Miller, A. Moore, Enabling open-source high speed network monitoring on NetFPGA, in: Network Operations and Management Symposium (NOMS), 2012 IEEE, 2012, pp. 1029–1035.
[19] L. Kekely, V. Pus, P. Benacek, J. Korenek, Trade-offs and progressive adoption of FPGA acceleration in network traffic monitoring, in: 2014 24th International Conference on Field Programmable Logic and Applications (FPL), 2014, pp. 1–4.
[20] Xilinx, Ultrascale Architecture and Product Overview, 2015.
[21] L. Filtr, Project WWW Page, 2010 http://l7-filter.sourceforge.net/.
[22] R.W. Floyd, J.D. Ullman, The compilation of regular expressions into integrated circuits, J. ACM 29 (3) (1982) 603–622.
[23] R. Sidhu, V.K. Prasanna, Fast regular expression matching using FPGAs, in: FCCM '01: Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, IEEE Computer Society, 2001, pp. 227–238.
[24] C. Clark, D. Schimmel, Efficient Reconfigurable Logic circuits for matching complex network intrusion detection patterns, in: 13th International Conference on Field Programmable Logic and Application, Lisbon, Portugal, 2003, pp. 956–959.
[25] C.R. Clark, D.E. Schimmel, Scalable pattern matching for high-speed networks, in: IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, CA, 2004, pp. 249–257.
[26] S. Yun, K. Lee, Optimization of regular expression pattern matching circuit using at-most two-hot encoding on FPGA, in: International Conference on Field Programmable Logic and Applications, 2010, pp. 40–43.
[27] I. Sourdis, J. Bispo, J.M.P. Cardoso, S. Vassiliadis, Regular expression matching in reconfigurable hardware, J. Signal Process. Syst. 51 (1) (2008) 99–121.
[28] C.-H. Lin, C.-T. Huang, C.-P. Jiang, S.-C. Chang, Optimization of pattern matching circuits for regular expression on fpga, IEEE Trans. Very Large Scale Integr. Syst. 15 (12) (2007) 1303–1310.
[29] B.C. Brodie, D.E. Taylor, R.K. Cytron, A scalable architecture for high-throughput regular expression pattern matching, SIGARCH Comput. Archit. News 34 (2) (2006) 191–202.
[30] T. Higuchi, T. Niwa, T. Tanaka, H. Iba, H. de Garis, T. Furuya, Evolving hardware with genetic learning: a first step towards building a darwin machine, in: Proc. of the 2nd International Conference on Simulated Adaptive Behaviour, MIT Press, 1993, pp. 417–424.
[31] L. Sekanina, Evolvable hardware, in: Handbook of Natural Computing, Springer Verlag, 2012, pp. 1657–1705.
[32] J.F. Miller, Cartesian Genetic Programming, Springer-Verlag, 2011.
[33] E. Stomeo, T. Kalganova, C. Lambert, Generalized disjunction decomposition for evolvable hardware, IEEE Trans. Syst. Man Cybern. B 36 (5) (2006) 1024–1043.
[34] A.P. Shanthi, R. Parthasarathi, Practical and scalable evolution of digital circuits, Appl. Soft Comput. 9 (2) (2009) 618–624.
[35] P. Kaufmann, C. Plessl, M. Platzner, EvoCaches: application-specific adaptation of cache mappings, in: Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS), IEEE Computer Society, 2009, pp. 11–18.
[36] Z. Vasicek, M. Bidlo, L. Sekanina, Evolution of efficient real-time non-linear image filters for FPGAs, Soft Comput. 17 (11) (2013) 2163–2180.
[37] S.M. Cheang, K.H. Lee, K.S. Leung, Applying genetic parallel programming to synthesize combinational logic circuits, IEEE Trans. Evol. Comput. 11 (4) (2007) 503–520.
[38] H. Esmaeilzadeh, A. Sampson, L. Ceze, D. Burger, Neural acceleration for general-purpose approximate programs, Commun. ACM 58 (1) (2015) 105–115.
[39] J.F. Miller, S.L. Smith, Redundancy and computational efficiency in Cartesian genetic programming, IEEE Trans. Evol. Comput. 10 (2) (2006) 167–174.
[40] B.W. Goldman, W.F. Punch, Analysis of Cartesian genetic programming's evolutionary mechanisms, IEEE Trans. Evol. Comput. 19 (3) (2015) 359–373.

Paper II

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

GROCHOL David and SEKANINA Lukas
In: *GECCO '16 Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*. New York, NY: Association for Computing Machinery, 2016, pp. 901-908. ISBN 978-1-4503-4206-3.

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

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

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

ABSTRACT

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

CCS Concepts

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

Keywords

Linear Genetic Programming, Network applications, Hash function

1. INTRODUCTION

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

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

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

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

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

GECCO '16, July 20-24, 2016, Denver, CO, USA

© 2016 ACM. ISBN 978-1-4503-4206-3/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2908812.2908825

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

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

2. RELATED WORK

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

2.1 Hash functions

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

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

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

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

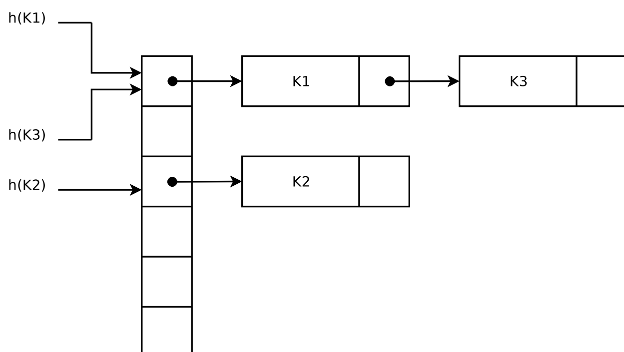


Figure 1: Hash table with separate chaining.

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

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

Figure 2: Example of LGP individual.

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

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

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

2.2 Linear Genetic Programming

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

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

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

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

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

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

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

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

2.3 Evolution of hash functions

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

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

3. HASH FUNCTION DESIGN

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

3.1 Towards fast hashing

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

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

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

3.2 Parallel LGP and its parameters

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

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

3.3 Initialization and fitness function

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

Table 1: LGP parameters

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

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

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

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

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

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

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

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

4. EXPERIMENTS AND RESULTS

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

4.1 Network Data

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

4.2 Hash functions used for comparison

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

4.3 Analysis of LGP Setting

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

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

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

Fig. 6 shows the number of instructions that were really

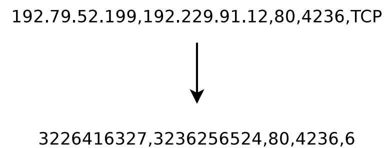


Figure 3: Example of conversion between a real network record and training vector.

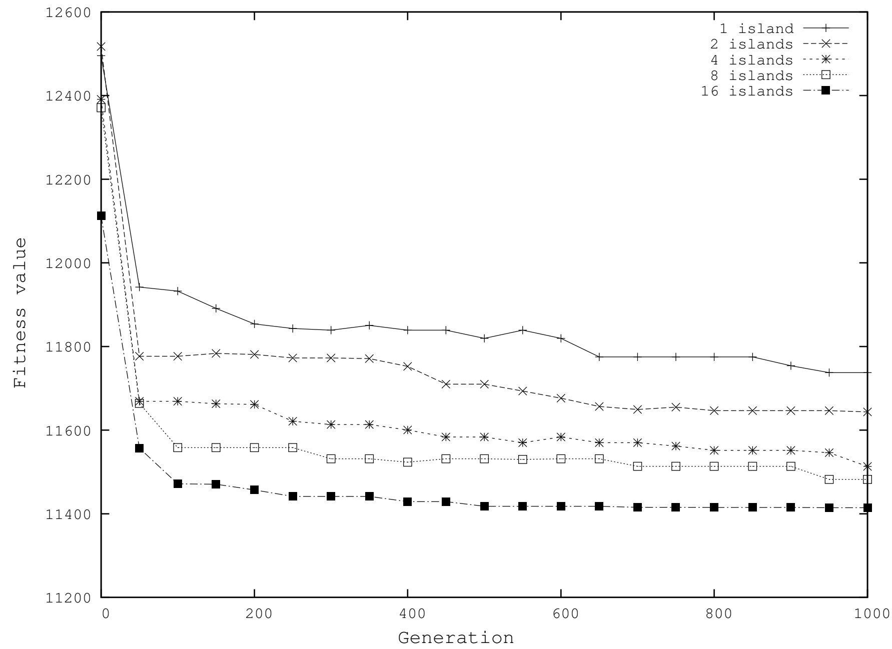


Figure 4: The progress of the best fitness score as median out of 20 independent runs on a different number of islands.

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

4.4 Evolved hash functions

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

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

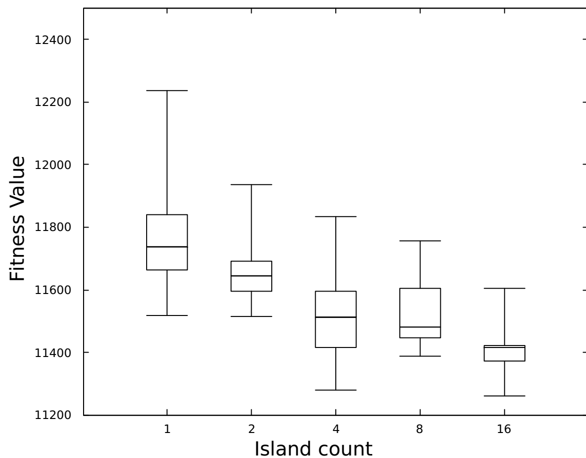


Figure 5: The best fitness values obtained from 20 independent runs on 1, 2, 4 , 8 and 16 islands.

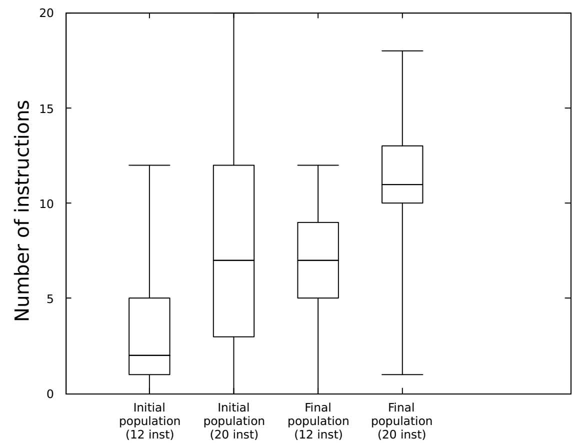


Figure 6: The number of instructions that were utilized in the initial population and final population if the program size is limited to 12 and 20 instructions.

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

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

Figure 7: Evolved hash function LGPHash1.

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

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

Figure 8: Evolved hash function LGPHash2.

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

4.5 Collision test

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

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

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

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

Figure 9: Evolved hash function LGPhashMult.

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

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

Figure 10: Evolved hash function LGPhash20inst.

Table 2: The number of collisions.

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

4.6 The execution time

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

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

Table 3: The average execution time.

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

Table 4: Overall quality of hash functions

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

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

4.7 Overall quality of hash functions

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

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

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

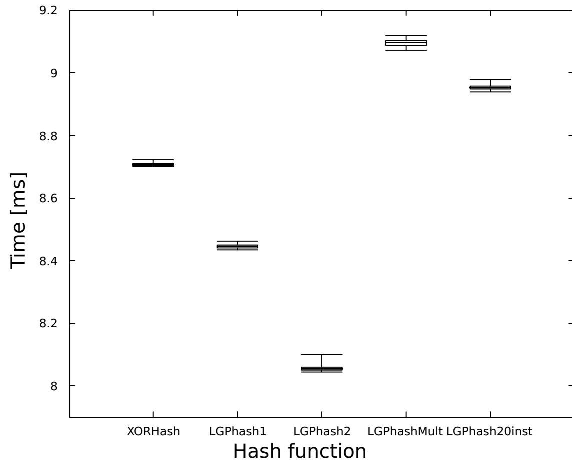


Figure 11: The execution time of selected hash functions on DataSet3 calculated from 20 runs.

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

5. CONCLUSIONS

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

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

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

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

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

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

Acknowledgment

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

6. REFERENCES

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

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

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

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

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

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

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

[8] R. Dobai and J. Korenek. Evolution of non-cryptographic hash function pairs for fpga-based network applications. In *2015 IEEE Symposium Series on Computational Intelligence*, pages 1214–1219. IEEE, 2015.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Paper III

Multiobjective Evolution of Hash Functions for High Speed Networks

GROCHOL David and SEKANINA Lukas
In: *Proceedings of the 2017 IEEE Congress on Evolutionary Computation*. San Sebastian: IEEE Computer Society, 2017, pp. 1533-1540. ISBN 978-1-5090-4600-3.

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 super-computer 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 ©2017 IEEE

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.

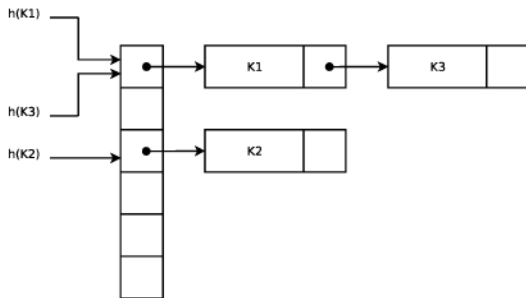


Fig. 1. Hash table with separate chaining.

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.

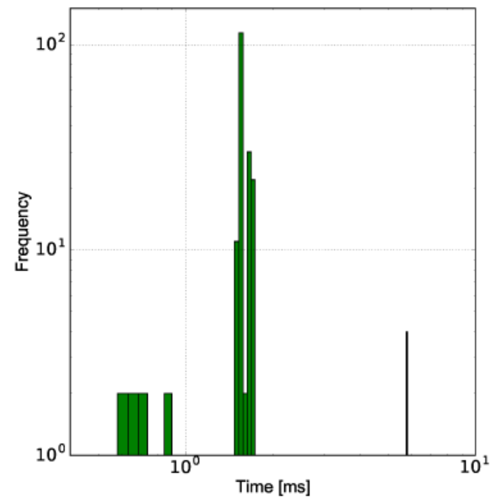


Fig. 3. The number of hash functions with a particular execution time in a 200 member LGP population.

(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
1 used-instructions = 0;
2 used-registers ← Insert(output-register);
3 while ⟨ i ← getLastInstruction(p) ⟩ do
4     if DestinationRegister(i) ∈ used-registers then
5         used-registers ← Insert(source-registers(i));
6         Increment(used-instructions);
7     remove instruction i from p;
8 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
1  used-instructions = 0;
2  r ← remove structurally redundant instructions from p
   using Alg.1;
3  M ← create matrix for instructions;
4  for i in r do
5      M ← using ASAP and ALAP routines to determine
       when i can be executed;
6  while ⟨ Some instruction(s) exist in M⟩ do
7      I ← find in M all instructions of the same type
       which can be executed together;
8      remove I from M;
9      increment(used-instructions);
10 return used-instructions;
```

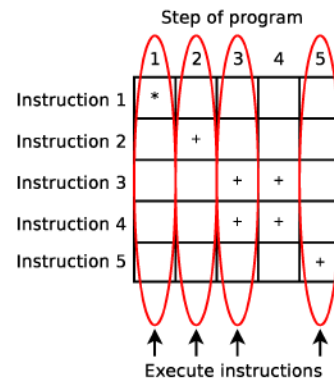


Fig. 4. Example of scheduling for a program given in Fig. 2. Instructions 3 and 4 can be executed together.

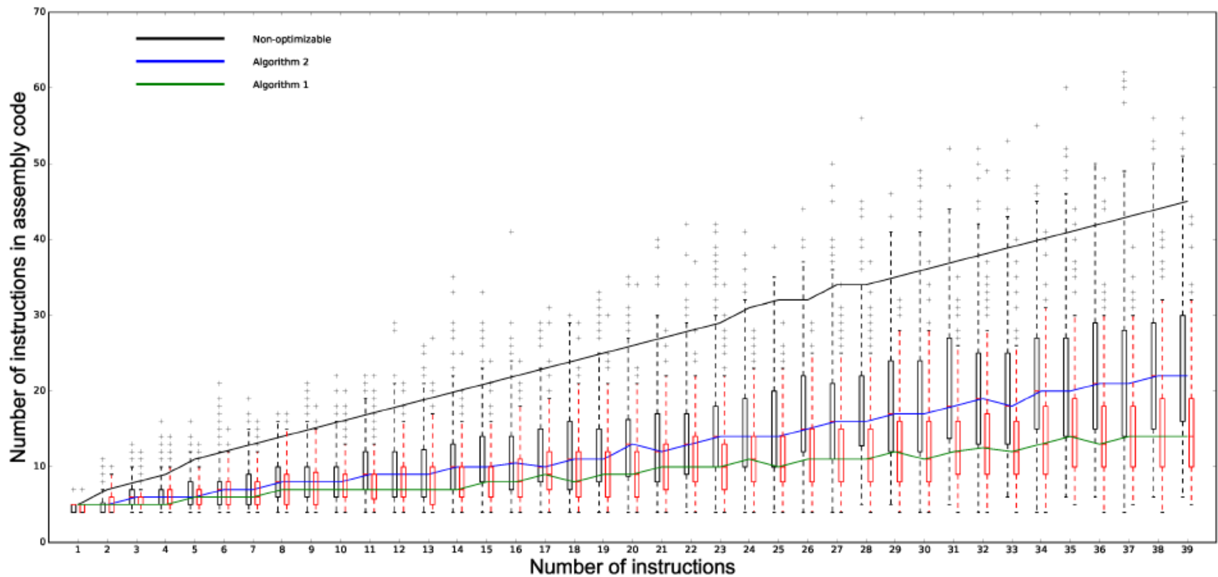


Fig. 5. The number of instructions in assembly code for 400 randomly generated programs containing 1 - 39 instructions, where the number of instructions was calculated according to Algorithm 1 (the median shown in green) and Algorithm 2 (the median is shown in blue). The size of non-optimizable programs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

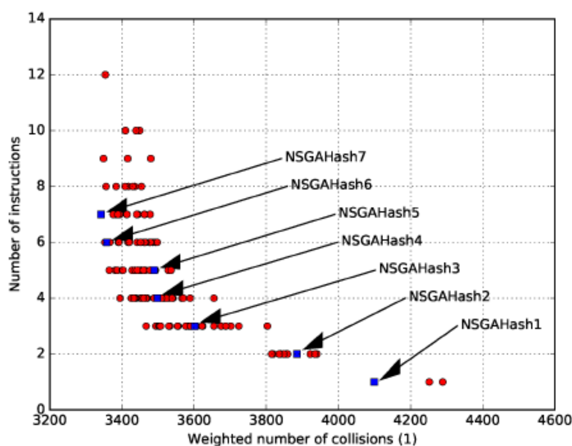


Fig. 7. Pareto fronts obtained from 30 independent runs. Selected hash functions (blue squares) are given in Fig. 6.

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

F. Overall Quality of Hash Functions

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

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

where b_j is the number of items assigned to j-th slot, m is the number of slots, and n is the total number of items. The

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.

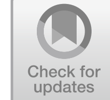
[27] "Secure hashing," http://csrc.nist.gov/groups/ST/toolkit/secure_hashing-.html, [ONLINE, accessed: 31. 1. 2016].

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

Paper IV

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

GROCHOL David and SEKANINA Lukas
In: *European Conference on Genetic Programming 2018*. Berlin: Springer International Publishing, LNCS 10781, 2018, pp. 187-202. ISBN 978-3-319-77553-1.



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.

© Springer International Publishing AG, part of Springer Nature 2018
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

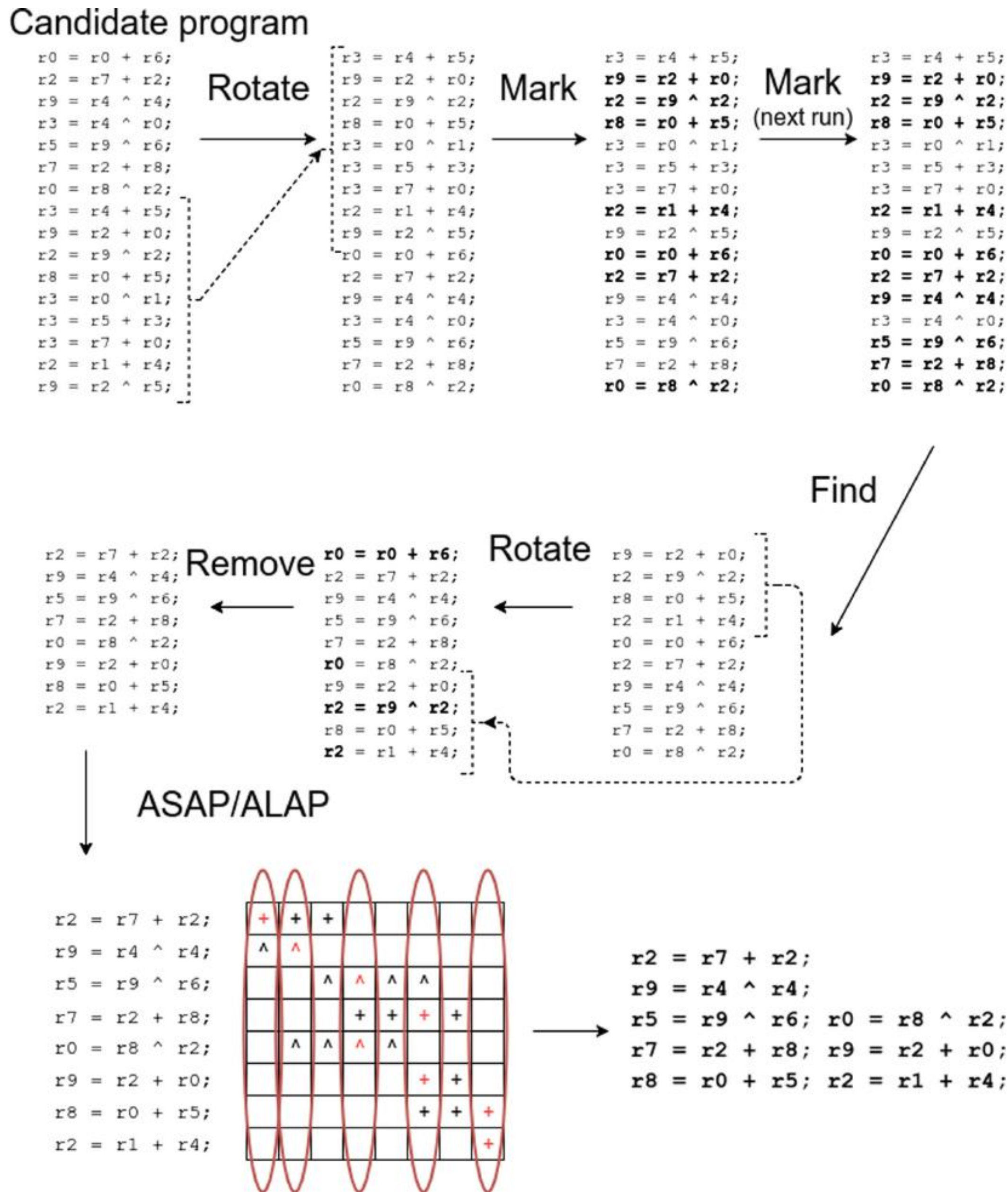


Fig. 2. Removal of unused instructions consists of rotating the candidate program to a configuration in which output register r0 is at position of the last instruction, identifying used instructions (in bold), removing unused instructions and rotating the code back. The optimized code is then scheduled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

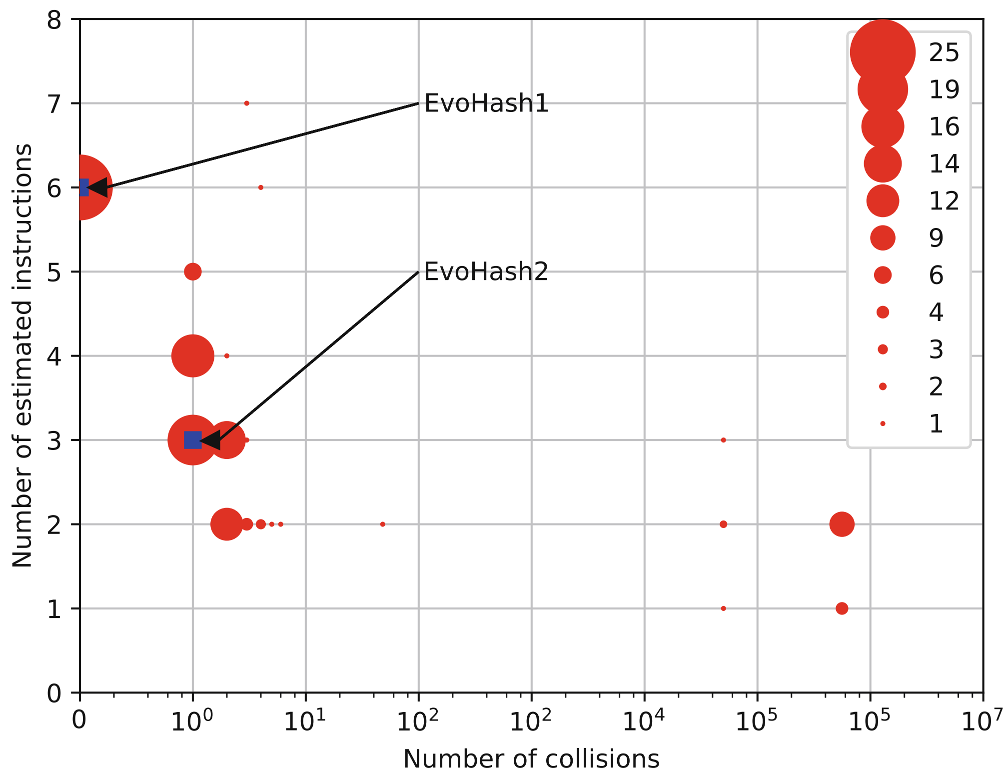


Fig. 4. Pareto fronts obtained from 100 independent runs of LGP. The size of the circle represents the number of identical solutions with the same properties. Selected hash functions (blue squares) are given in Fig. 3. (Color figure online)

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

Paper V

Fast Reconfigurable Hash Functions for Network Flow Hashing in FPGAs

GROCHOL David and SEKANINA Lukas
In: *Proceedings of the 2018 NASA/ESA Conference on Adaptive Hardware and Systems.* Edinburgh: Institute of Electrical and Electronics Engineers, 2018, pp. 257-263. ISBN 978-1-5386-7753-7.

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.

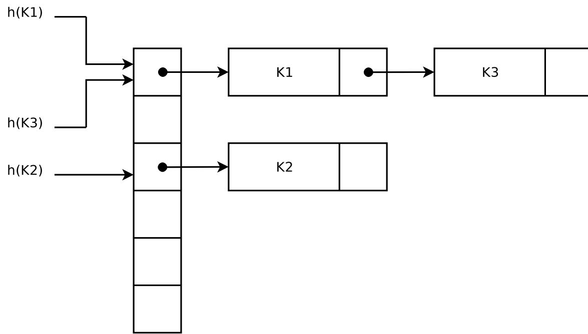


Fig. 1. Hash table with separate chaining.

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:

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

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

and s is the number of memory slots. This function 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.

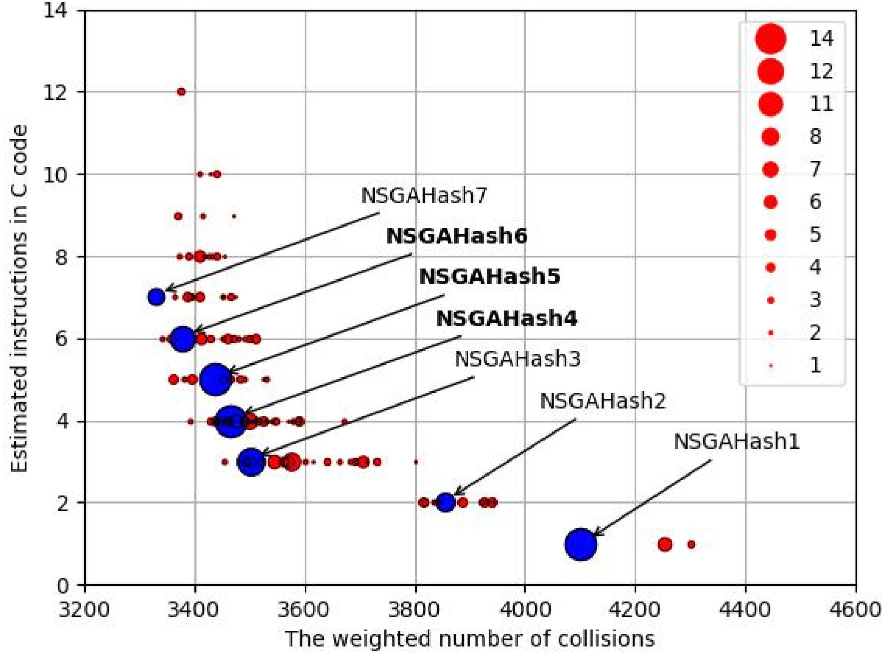
A. FPGA implementation

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.

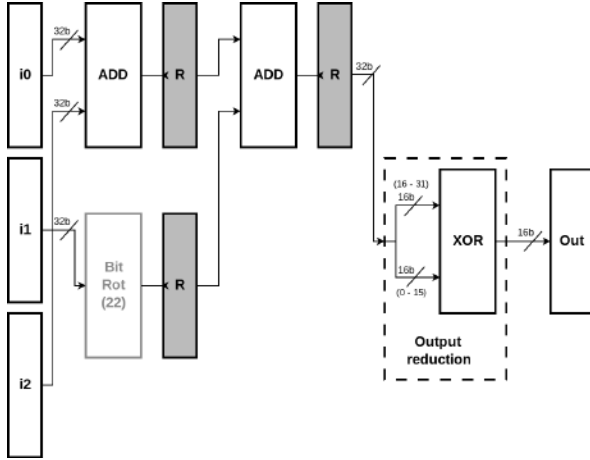


Fig. 3. Pipelined implementation of NSGAHash4.

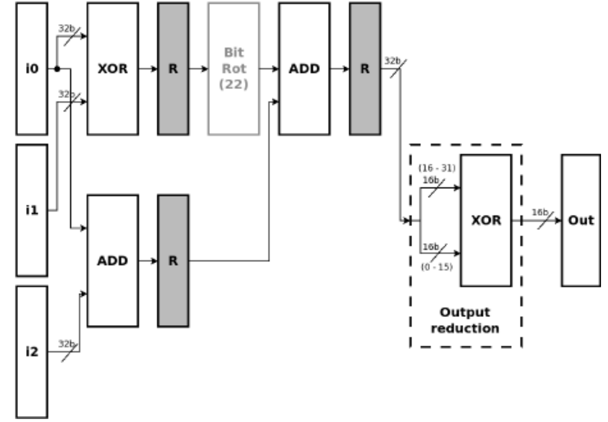


Fig. 4. Pipelined implementation of NSGAHash5.

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

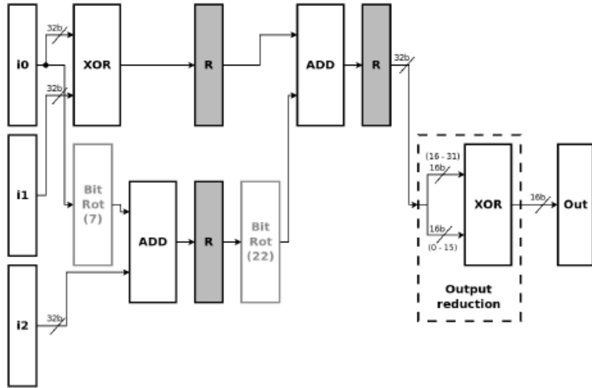


Fig. 5. Pipelined implementation of NSGAHash6.

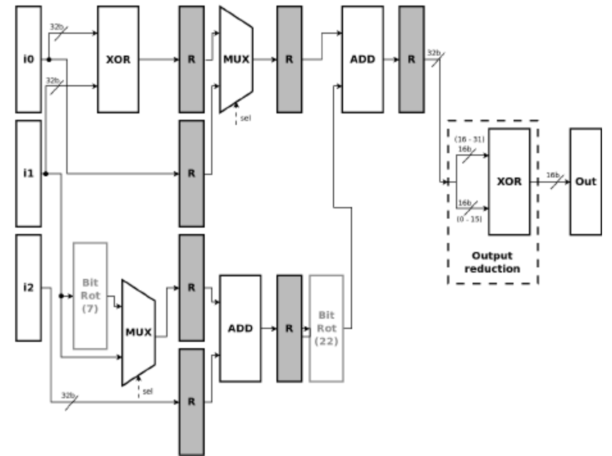


Fig. 6. Reconfigurable hash function RecoHash.

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

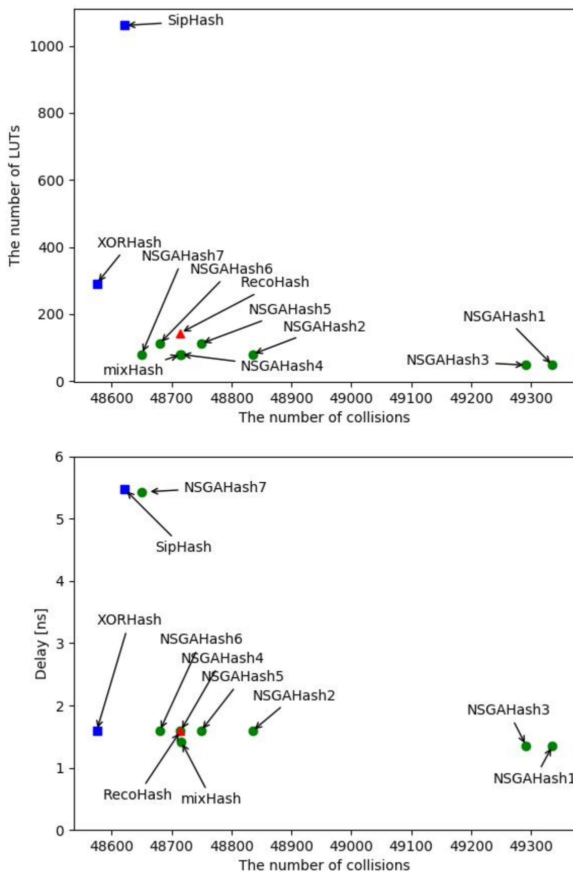


Fig. 7. Parameters of implementations of hash functions in Virtex-7. The number of collisions is given for DataSet3.

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.