Rapid Evolution of Time-Efficient Packet Classifiers

Ralf Salomon, Harald Widiger, and Andreas Tockhorn
Faculty of Computer Science and Electrical Engineering
University of Rostock, 18051 Rostock, Germany
Email: {ralf.salomon,harald.widiger,andreas.tockhorn}@uni-rostock.de

***Abstract*— Communication networks today are facing an ever increasing network traffic as well as raising quality-of-service agreements, which together demand for high performance network *routers*. Since a router has to search a large set or routing rules for every incoming packet, it normally utilizes efficient search mechanisms, such as trees or hash tables. This paper evolves hash functions directly in hardware and also discusses an improved initialization process. On a benchmark test consisting of 65,536 routing rules, the final hash functions consume an average of about 1.3 memory accesses per incoming data packet.**

I. Introduction

As is well known, *routers* are essential components in almost any digital communication network, since they control all the network traffic. A router's main task is, as Figure 1 indicates, to route incoming data packets from input lines to proper output lines. In the pertinent literature [14], the input/output lines are also referred to as *ports*. A packet consists of two parts, its header and its actual (user) data content. In the simplest case, a router makes its routing decisions based on some particular header information fields, such as the source and destination MAC addresses, the source and destination IP addresses, and/or the utilized transport protocol (UDP or TCP). In more complex situations, the router may also consider some of the packet's user data content.

The routing task is not always as easy as it appears at first glance. It might happen, for example, that a single output port or even the entire router is congested. In such situations, the router cannot forward all incoming packets to their destinations; rather, it has to drop selected packets. This decision may depend on the packet's actual content and/or on the quality-of-service negotiated for selected connections. For example, dropping a small number of voice-over-IP packets might help the router to overcome its congestion. It might even happen that a router is forced to manipulate packets for example, when inserting MPLS label stacks into packets [17], In summary, a router processes every incoming packet and forwards (routes) it to the desired port, which depends on the packet's content, particularly the packet's header fields. This problem is also known as the packet classification problem. In addition, increasing numbers of input/output ports and raising bandwidths on each port as well as increasing quality-of-service demands request routers to process incoming packets as fast as possible. Thus, designers have to construct routers with very low latencies, in order to assure traffic and quality-of-service demands, for example, in a voice-over-IP connection.

Figure 2 illustrates a straight-forward approach in which a router maintains a potentially large data base, which defines how to route incoming packets. A single rule of such a data base can be formulated like: `if ( packet.dest_IP equals 130.60.48.8 ) then route-to-port_5`. Furthermore, a rule might contain wilde cards and/or ranges so that it applies to many different packets keeping the data base's size limited. In the remainder of this paper, the packet content that is responsible for selecting a rule is referred to as the *key*.

Conversely, for each incoming packet, the router has to search its data base until a rule matches, and then executes the specified action(s), i.e., packet content manipulations and routing to the predefined output port. With n denoting the

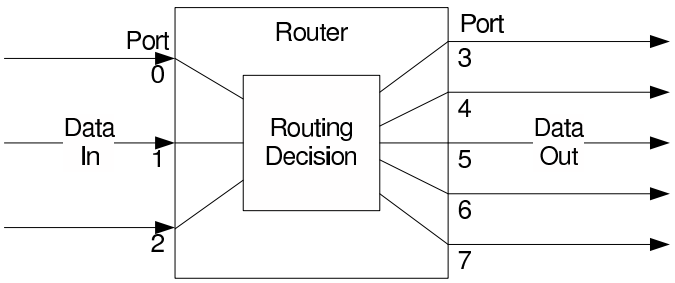

Fig. 1. The main task of a router is to control the network traffic by forwarding incoming packets to the correct destinations.

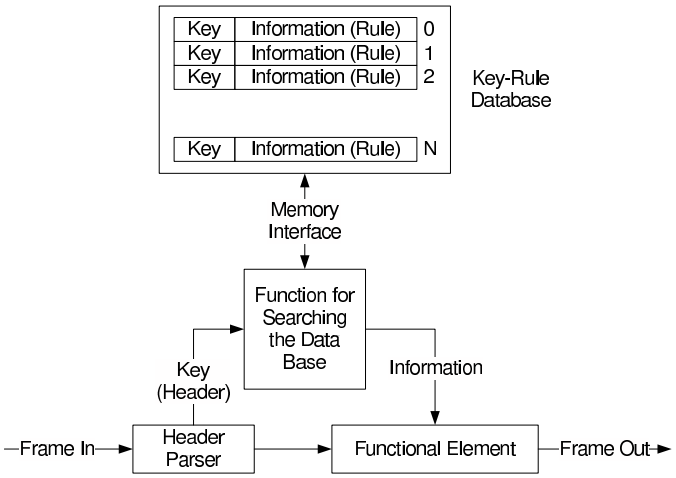

Fig. 2. A packet classifier utilizes a data base to map every incoming packet onto the proper output port.

number of rules in the data base, a sequential search through the data base requires an average of $n/2$ comparisons resulting in a search complexity of $O(n)$. With a steady increase in both the bandwidth and the number of ports, the size of the data base and thus the number of data base lookups are also steadily increasing. With a data base size of up to 160,000 [10] in state-of-the-art routers such a sequential search might become too expensive in terms of resulting latencies. In other words, the mechanisms for finding the correct routing rule devotes particular emphasis.

The concept of hash tables offers a powerful search mechanism, since they might yield a constant time complexity $O(1) \ll O(n)$ under certain circumstances [9]. Unfortunately, the routing profile of a single router changes over time, which would require adaptive hash tables in order to operate time-efficient. Accordingly, previous research [16] provides a proof-of-concept that hash tables can be directly employed in hardware and that the required online adaptation can be achieved by an evolutionary algorithm, which can also be directly employed in the very same hardware. Therefore, this intrinsic (evolvable) hardware approach yields both mostly constant search time $O(1)$ and constant online adaptation to changing routing profiles. Since this paper investigates the behavior of various genetic algorithms, Section II provides a detailed description of the developed hardware platform. Other approaches, such as binary trees and binary search in ordered lists, would require $O(\lg n)$ memory accesses on average, which would be better than a linear search but worse than hashing; further computational costs for inserting and deleting rules are not considered here.

Since the hardware platform has been developed in previous research, this paper focuses on how to efficiently evolve such hash functions in hardware. To this end, Section III describes the experimental setup in full detail. The results, as presented in Section IV, indicate that genetic algorithms can efficiently evolve reasonably-good hash functions within about 200 generations.

In order to speed up the initial evolutionary process, which starts off with randomly generated individuals, Section V describes an initialization process, which extracts certain statistical properties form the initial rule set. The results indicate that this initialization significantly improves both the evolutionary process and the final solution. Finally, Section VI concludes with a brief discussion.

II. Adaptive Hashing directly in Hardware

This section describes the problem in more detail and also provides a brief overview of previous research. The presentation starts off with a brief description of hash functions and their properties.

A. Hash Functions: Construction and Properties

In general, a search algorithm of any kind is required in cases where the domain is much larger than the elements to be stored, and/or where the domain size exceeds the available memory capacity. Assume, for example, an algorithm stores 100 different 16-bit integer values. Then the domain would consist of 65,536 different values, and thus, a memory of $2\times65,536$ bytes would be way too excessive to merely store 100 integer values; more than 99 % of the memory would not be used at all.

A hash function $h(x)$ maps a value x onto a hash value, which is usually from a much smaller domain $\{h(x)\} \ll \{x\}$ than the argument domain $\{x\}$. Assume, for example, a packet classifier (router) with 2^{ee}=1024 rules and packets with 32-bit wide keys, which could represent the destination IP addresses. Then, the hash function has to map 4,294,967,296 different values onto a new domain with 1024 entries. In order to work efficiently, the actual number of rules would be less than or equal to 1024.

Since a hash function maps values from a large domain onto a much smaller one, not all different values can have different hash values. That is, it occurs that two hash values $h(x) = h(y \neq x)$ are equivalent even though their arguments are not. In a practical application, such *collisions* must be resolved. This can either be done by rehashing $g(h(x))$ the hash value by another hash function $g()$ or by searching for a free memory entry. Such a (linear) search can be done by adding a constant prime number, including the value 1, to the hash value.

For a given set of values, the quality of a hash function can be measured by the number of conflicts that occur when hashing all given keys into memory. A hash function that maps all *given* values onto different hash values, i.e., memory entries, is called perfect; in practical applications the number of collisions does not vanish. The reason for this is that the actual values be mapped are not known in advance.

The optimization task is thus to find a particular hash function $h(x)$ that maps all given n input values $x_{1..n}$ with as few as possible conflicts. Whether or not the number of conflicts vanishes depends on both the arguments and operators that can be employed into the hash function.

B. Routing using Hash Functions

Figure 3 sketches the evolvable hardware platform that has been developed in previous research [16]. The hardware works as follows: A key parser extracts the key, i.e., the destination IP address, from an incoming packet, and by means of a switch, forwards it to the hash function that is entirely realized in hardware. The hash function maps the key onto the classification rule, which is consequently forwarded to the actual routing unit (shown only in Figure 2). Because the hash function also has to resolve conflicts, it always compares its input key with that stored along each rule. And in case of a collision, the hash function linearly searches the memory, as has already been described in Subsection II-A.

The hardware platform shown in Figure 4 also features a second hash function, which allows for online updates, and thus, evolution in hardware. The hardware evolution model (bottom part of the figure) can apply variations to the second hash function and can also monitor the performance, i.e., number of conflicts, of both. Depending on the actual performance,

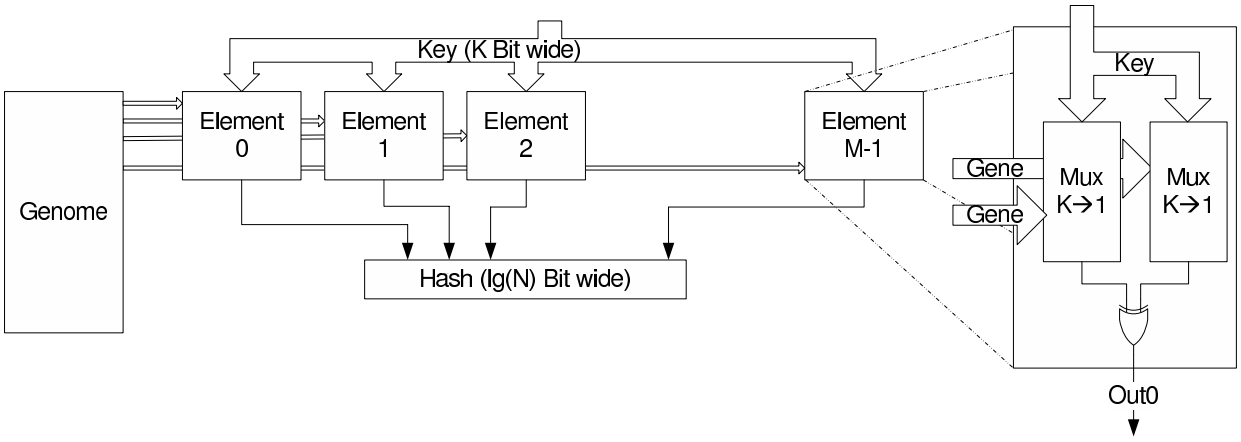

Fig. 3. This hardware classifier platform has been developed in previous research [16]. A genome feeds $\lg n$ elements, which select two bit positions from k-bit-wide keys by a number of multiplexers.

the platform can utilize either of the hash functions by properly configuring the two switches.

Each of the two hash functions are defined by a bit string S consisting of $s = 2\lg(k)\lg(n)$ bits, with k denoting the number bits to code the input values x and n denoting the number of bits to code the hash values $h(x)$. In the example presented above, the values were $k = 32$ and $n = 1024$. Thus, the hash function uses two times $\lg k$ bits to select a bit position of the input value for each of the $\lg n$ bits that code for the hash values.

For the evolutionary algorithm, the task is to find an optimum in a search space with $s = 2\lg(k)\lg(n)$ dimensions, which is $s = 2 \times 5 \times 10 = 100$ in the example discussed above. For the interested reader, the Subsection II-C explains how this configuration is done in hardware.

The hardware platform as described above realizes all operations in hardware, so that no software is involved at any place. Thus, this packet classifier operates at a very high speed given that the hash function is properly evolved.

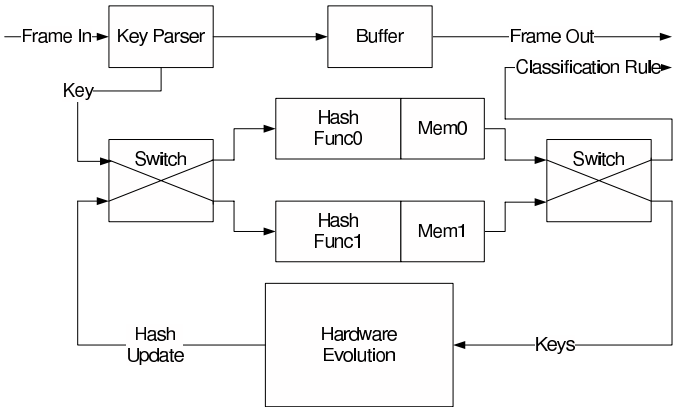

Fig. 4. The implementation of the hash function in hardware. For details, please, see text.

C. Realizing Hash Functions in Hardware

For the reader interested in evolvable hardware, it might be worth describing some implementation details. The implementation is based on a field-programmable-gate-array (FPGA). The genome is fed to $\lg n$ equivalent elements. Each single element utilizes $2\lg k$ bits to freely select two arbitrary bits from the input key (using multiplexers denoted as *Mux* in the figure). These two arbitrarily selected bits are then processed by an exclusive-or gate, thus providing one single bit to the hash function.

In this particular implementation, the hash function consists of $\lg k$ pairs of exclusively ORed (XOR for short) input bits arbitrarily chosen from the packet's key. This way, the system can realize $2^{2\lg(k)\lg(n)}$ different hash functions. Thus, the optimization goal for the application at hand is to find the best one in a search space consisting of $2^{2\lg(k)\lg(n)}$=2^{100}. For further implementation details, the interested reader[1] is referred to the literature [16].

III. Algorithms and Methods

This paper employs genetic algorithms to evolve hash functions for the packet classification problem. Genetic algorithms are a member of the class of heuristic population-based search procedures known as *evolutionary algorithms* that incorporate random variation and selection. Evolutionary algorithms provides a framework that mainly consists of genetic algorithms [6], evolutionary programming [5], [4], and evolution strategies [11], [13].

A genetic algorithm maintains a population of μ individuals, also called parents. In each generation, it generates λ offspring by copying randomly selected parents and applying variation operators, such as mutation and recombination. It then assigns a fitness value (defined by a fitness or objective function) to each offspring. Depending on their fitness, each offspring is given a specific survival probability.

[1]VHDL code can be directly received by sending an email to Harald Widiger.

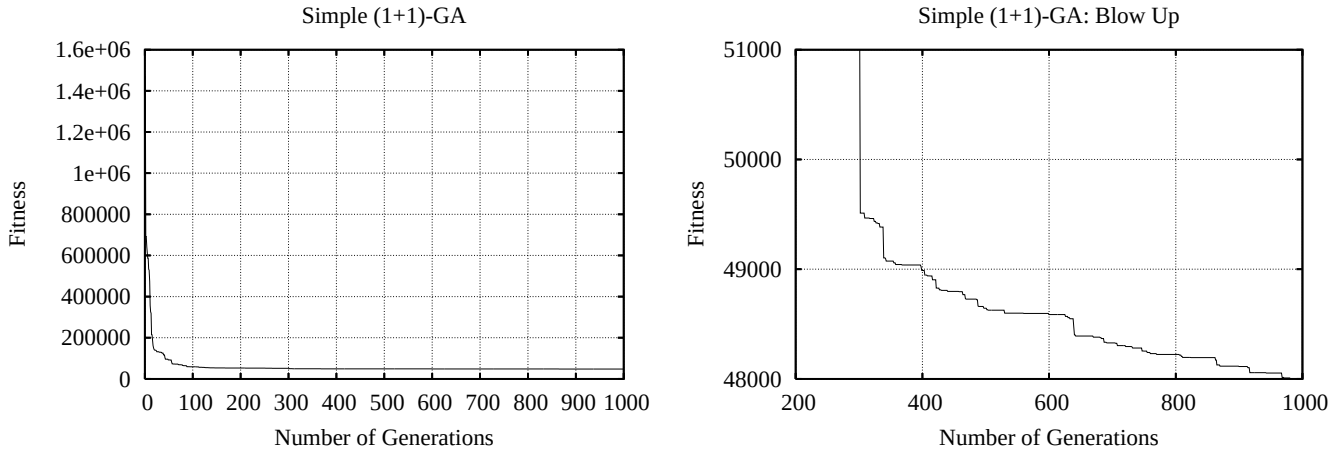

Fig. 5. The behavior of a simple (1+1)-GA; the left-hand-side shows a blow up in order to present more details.

Since the problem at hand is already encoded in a bit string S consisting of $s = 2\lg(k)\lg(n)$ bits, this paper directly uses that bit string as the genome. The mutation operator flips every bit randomly with a mutation probability of $p_m = 1/s$. Because the bit representation is not given in any particular ordering, this paper also applies uniform recombination, which exchanges corresponding bits of two randomly selected parents with a probability p_r=0.5.

Depending on the selection scheme, the algorithms are either denoted as (μ,λ)-GA or (μ+λ)-GA for short. The first selection scheme indicates that it choses the parents for the next generation from the offspring only, whereas the second one also considers the parents from the current one. When using the a (μ,λ)-GA, this paper also considers the best parent for selection, also known as elitism, in order to avoid any deterioration.

In order to achieve a permanent online adaptation, the *hardware* implementation of the genetic algorithm is an integral part of the entire system. Thus, the hardware is restricted to rather small population sizes, such as a standard (1+1)-GA or a (1,6)-GA; larger population sizes would simply require too much hardware resources and/or computation time. However, this paper also presents some benchmark tests for comparison purposes.

Since the goal of the optimization procss is to evolve a hash function with as few conflicts as possible, the fitness function f is the sum of all conflicts. For the evaluation, this paper uses a hash table with 65,365 entries (i.e., $\lg n$=16-bit wide table indices), and draws 32,768 keys with a width of $k = 32$ bit at random. The fitness function then inserts the 32,768 keys one after the other into the hash table, and in so doing, counts the number of conflicts.

The hardware platform as described above has already been realized, and has been used as a proof of concept. For the investigation presented here, this paper has to resort to simulations and randomly generated keys, since no adequate and/or representative benchmarks exisits; neither do the authors have access to a representative network in which the existing hardware platform could be incorporated. It may be mentioned here that the resulting execution time is the only noteable difference between the simulation and the actual hardware platform. Since the simulation model is implemented in SystemC, it mimics the actual hardware implementation accurately. One single run with a $\lambda = 6$ over 1000 generations require approximately 30 minutes of simulation time.

IV. Results

Figure 5 shows the behavior of a simple (1+1)-GA averaged over 10 independent runs. As can be seen on the left-hand-side, the hash function starts off with about 850,000 conflicts. But then, the procedure rapidly arrives at a value of about 48,300 conflicts, as can be seen on the right-hand-side of Figure 5, which shows a blow up. A final value of about 48,300 conflicts results in an average of slightly less than 2.5 memory access per key. This value is reasonably good and significantly better than previous research has achieved [7].

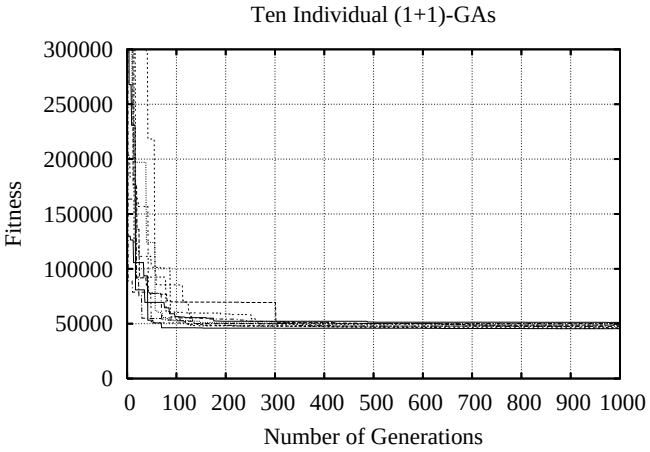

Fig. 6. This figure shows ten independent runs, which all show very similar behavior. Thus, the remainder of this paper presents only averaged performance figures.

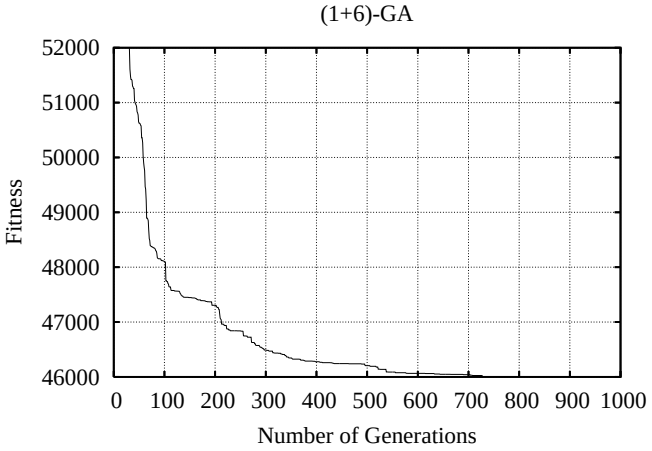

Fig. 7. The behavior of a simple (1,6)-GA.

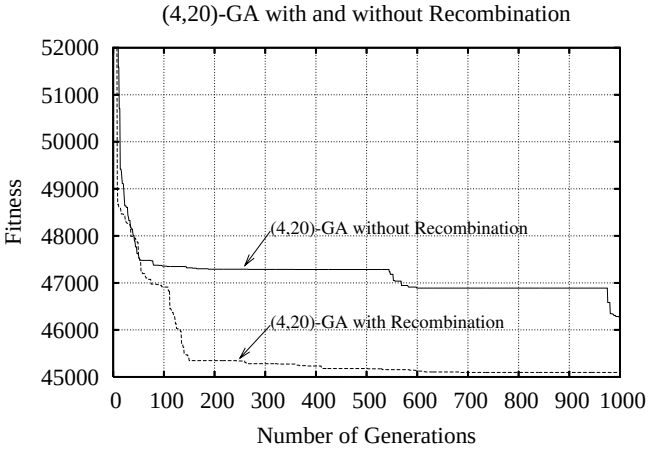

Fig. 9. The behavior of a (4,20)-GA with and without recombination.

In addition, Figure 6 shows ten individual runs. As can be clearly seen, all runs exhibit a very similar behavior. It is thus reasonable that the remainder of this section discusses only averaged performance figures.

Figures 7 to 9 show how larger population sizes affect the algorithm's performance. In general, as can be expected, larger population sizes accelerate the evolutionary process. However, neither the employment of larger population sizes nor the usage of recombination is able to significantly improve the final fitness value.

As can be expected, Figures 8 and 9 both indicate that recombination significantly improves the convergence speed of a genetic algorithm. Even though larger population sizes operate faster, they normally exhibit an inferior *sequential* performance. But since the system described here is be used in hardware, the extra effort in terms of hardware resources does not seem worth it, especially since this would accelerate only the initial stage (see, also, Section V). Once the system has converged to a suitable solution, even a simple (1+1)-GA adapts to changes of the rule set sufficiently fast.

Finally, this section discusses the scaling behavior of this approach with respect to the number of keys. To this end, Figure 10 shows how often the evolved hash function accesses the memory on average for each incoming packet for $8\,k$, $16\,K$, $32\,K$, and $64\,K$ rules. It can be seen that the average value is between 2.4 and 3.2.

V. Fast Bootstrapping by Smart Initialization

A. The Initialization Algorithm

Section IV has presented the results when evolving hash functions by means of (μ,λ)-genetic algorithms. It is surprinsing that with an average of 1.5 conflicts (i.e., 2.5 memory accesses), these rather canonical algorithms yield results better than those reported in the literature [7]. Furthermore, the results indicate a significant improvement from the randomly chosen starting points to the final fitness values. This section investigates to what extent a modified initialization procedure

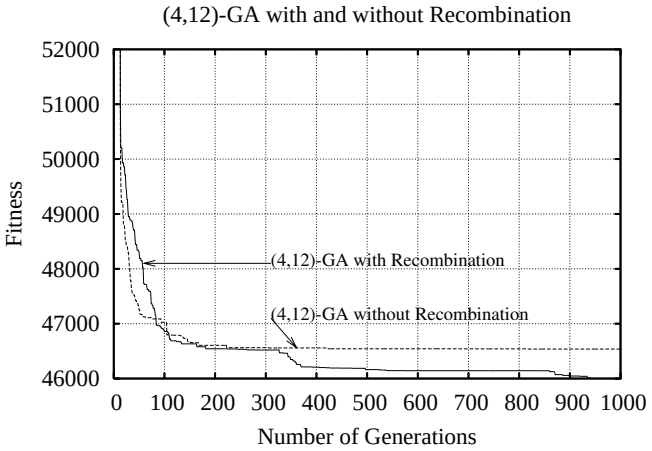

Fig. 8. The behavior of a (4,12)-GA with and without recombination.

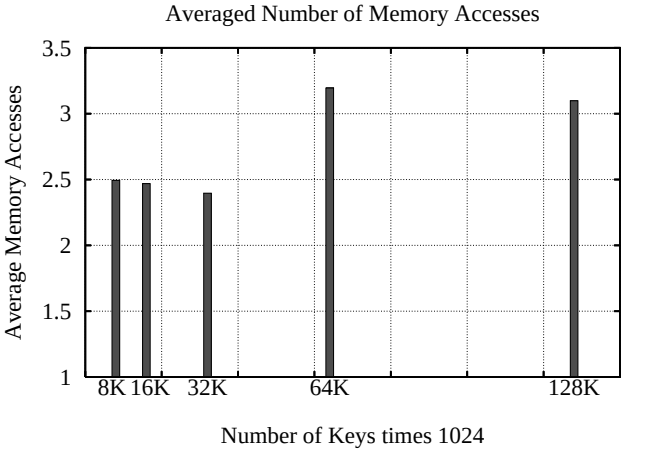

Fig. 10. This figure shows how often the evolved hash function accesses the memory on average per incoming packet for $8\,k$, $16\,K$, $32\,K$, $64\,K$, and $128\,K$ rules.

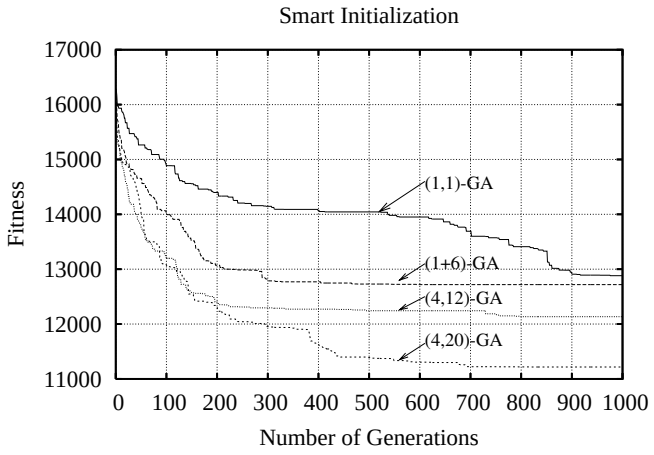

Fig. 11. The performance of various genetic algorithms on the evolution of a hash function with smart initialization. With smart initialization, both the starting and final fitness values are about four times better than with random initialization.

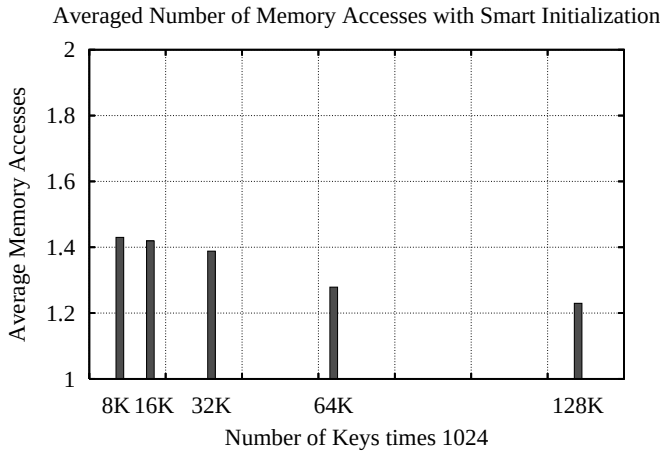

Fig. 12. This figure shows how often the evolved hash function accesses the memory on average per incoming packet for $8\,k$, $16\,K$, $32\,K$, $64\,K$, and $128\,K$ rules.

would be able to improve the algorithm's performance with respect to both runtime and final fitness values. A better start-off in the first phase, would help the router particularly in its start-up phase, also called bootstrapping, since during that stage, a non-optimized hash function cannot be expected to perform well. Therefore, this section aims at a smart initialization process, and abandons random initializations; it rather initializes the parents of the first generation in dependence of some properties of the initial data base's rule set. To this end, this section adopts the following strategy: for all k bits of the (rule) keys, it counts how often the kth bit position is set to "1". It then selects those $2\lg n$ bits for the hash value that are closest to half of the total number of rules. That is, the initial hash function utilizes those bits that are statistically closest to 50 % be set; the assumption is that those bits have the highest entropy.

B. Results

Figure 11 shows the performance of the different genetic algorithms when using the initialization procedure described above. First of all, the procedures starts off at an initial value of about 16,000 conflicts, which is way better than the best result obtained with a random initialization. The drastic improvement suggests that the results reported in Sections IV constitute local optima. Second, the genetic algorithms more or less all arrive at a final value of about 11,500 conflicts. With 32,768 keys (data base rules), the hash function accesses the memory about 1.35 times on average.

Figure 12 shows how often the evolved hash function accesses the memory on average per incoming packet for $8\,k$, $16\,K$, $32\,K$, and $64\,K$ rules. It can be seen that the average value is between 1.35 and 1.44, with even *decreasing* numbers for larger rule sets.

C. Realizing the Initialization in Hardware

This initialization procedure has been implemented in hardware as follows (see, also, Figure 13): the initialization module employs k counters, one for each bit of the rule key. Then, for every key a finite-state machine increments or decrements all counters if the corresponding key bit is "1" or "0", respectively. After all keys have been processed, each counter containes the difference between the occurrences of "1"s and "0"s at the corresponding positions. For this stage, the hardware requires k counters with a witdh of $\lceil \lg n + 1 \rceil$ bits.

In a second stage, the initialization module performs a bubble sort on the counter values by means of another finite-state machine. To this end, the module calculates the 2-complement if a counter value is negative. Then, the module employs k counters with a witdh of $\lceil \lg n + 1 \rceil$ bits. Since the hash value has $\lg(n)$ bit positions, each of which is an XOR of two bit from the key, the initialization procedure selects the $2\lg n$ best bit positions of the key, i.e., those bit positions where the occurrences of "0" and "1" is as equal as possible. Out of this selection, the procedure, XORs the best with the

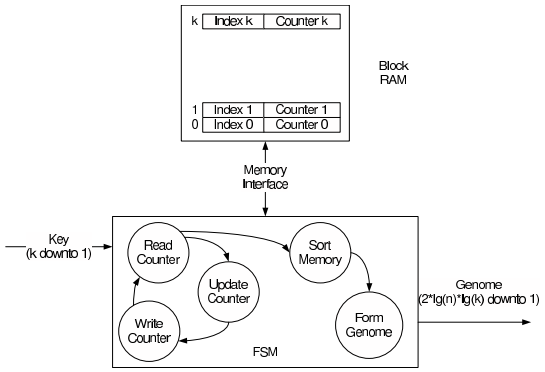

Fig. 13. The realization of the finite-state machine for calculating proper initialization values from the initial rule data base. For details, please, see text.

worst, the second best with the second worst, and so on. The resulting genome serves as a seed value for the subsequent evolutionary process.

Even though this paper has used simulations in order to obtain benchmark tests, the evolution of a hash function has also been implemented in an FPGA-based hardware. In order to keep the hardware resources at a minimum, the smart initialization algorithm holds all counters in a single RAM Block (BRAM) provided by FPGAs today. The capacity of one BRAM suffices for 2^{24} 512-bit wide keys. The counters hence do not consume any of the logic resources. Only the two finite-state machine require a descent number of logic elements. The current implementation requires an area of 206 slices (logic elements in a Xilinx FPGA), which increases the size of the entire packet classifier by about 7 %. In light of the achieved performance, this extra hardware burden seems worth it.

VI. Conclusions

This paper has applied various genetic algorithms to evolve hash functions directly in hardware in order to address the problem of fast packet classification in state-of-the-art network routers. It turned out that simple (1+1)-GAs evolve hash functions reasonably fast.

In order to speedup the behavior in the very first generations, e.g., after switch-on or after any hardware reset, this paper has also investigated an initialization procedure that directly exploits properties of the given rule set. This initialization requires only one complete sweep through all rules of the data base, and thus compares to one single fitness evaluation.

The hash function, the genetic algorithm, the data path of the packet classifier, and the additional initialization procedure have all been implemented in hardware using the VHDL description language. In a Xilinx Virtex4-FX20 FPGA [18], the system consumes 3000 slices of logic and runs at a clock rate of more than 125 MHz. At this speed, the classifier is capable of performing more than 60 million classifications per second, when assuming two memory accesses per classification (indicated by the simulation results). Thus, a single packet classifier would be theoretically able to process 28.8 GBit/s of Ethernet traffic at wire speed, even if the packets have the minimal size.

Further research will be dedicated to an analysis of why the proposed initialization procedure yields such a high performance improvement. It can be expected, of course, that a proper initialization significantly accelerates the evolutionary process. However, it is yet unclear, why the genetic algorithms converged at about 45,000 conflicts when starting with a randomly initialized population, and were not able at all to achieve a value of about 16,000 conflicts already from the very beginning.

Future research will also be dedicated to an evaluation of the proposed packet classifier system in a real-world communication network. To this end, a project is set up with the industrial research partner.

Finally, the packet classifier could benefit from a further speed up of the evolutionary process, since it would allow for faster adaptations to changing rule data bases. To this end, future research will be evaluating different collision resolution as well as potential benefits of the utilization of memory interleaving methods. Furthermore, future research will be investigating modified evaluation models as well as the parallelization of the evolutionary process.

Acknowledgements

The authors gratefulle thank the Simens Communications Corp., Greifswald, for bringing the problem to the authors' attention as well as for their valuable feedback on various technical details. Part of this research has been supported by the 4th Regional Research Priority Program (LFS) on information and communication technologies sponsored by the European Union, grant number UR0202510/2005.

References

[1] T. Bäck, U. Hammel, and H.-P. Schwefel, Evolutionary Computation: Comments on the History and Current State. *IEEE Transactions on Evolutionary Computation*, **1**(1):3-17, 1997.

[2] A. Broder and M. Mitzenmacher, Using Multiple Hash Functions to Improve IP Lookups, in *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001)*, pp. 1454-1463, 2001.

[3] E. Damiani and A.G.B. Tettamanzi, On-Line Evolution of FPGA-Bases Circuits: A Case Study on Hash Functions, in *Proceedings of the first NASA/DoD Workshop on Evolvable Hardware*, pp. 36-33, 1999.

[4] D.B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Learning Intelligence.*. IEEE Press, NJ, 1995.

[5] L.J. Fogel, Autonomous Automata. *Industrial Research*, **4**:14-19, 1962.

[6] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, Reading, MA, 1989.

[7] P. Gupta and N. McKeown, Packet Classification using Hierarchical Intelligent Cuttings. *IEEE Micro*, **20**(1):34-41, 2000.

[8] P. Gupta and N. McKeown, Algorithms for Packet Classification, in *IEEE Network*, **15**(2):24-32, 2001.

[9] D.E. Knuth, *The art of computer programmingm, vol. 3, sorting and searching.* Addison-Wesley, 3rd edition, 1998.

[10] X. Meng, Z. Xu, B. Zhang, G. Huston, S. Lu, and L. Zhang, IPv4 Address Allocation and the BGP Routing Table Evolution, *ACM SIGCOMM Computer Communication Review (CCR)*, Special Issue on Internet Vital Statistics, **35**(1):71-80, 2005.

[11] I. Rechenberg, *Evolutionsstrategie* (Frommann-Holzboog, Stuttgart, 1994).

[12] R. Salomon. Reevaluating Genetic Algorithm Performance under Coordinate Rotation of Benchmark Functions; A survey of some theoretical and practical aspects of genetic algorithms. *BioSystems*, **39**(3):263-278, 1996.

[13] H.-P. Schwefel. *Evolution and Optimum Seeking.* John Wiley and Sons, NY. 1995.

[14] A. Tannenbaum, *Modern Operating Systems.* Prentice Hall, 2001.

[15] G. Tufte and P.C. Haddow, Prototyping a GA Pipeline for Complete Hardware Evolution, in *Proceedings of the first NASA/DoD Workshop on Evolvable Hardware*, pp. 143-150, 1999.

[16] H. Widiger, R. Salomon, and D. Timmermann, Packet Classification with Evolvable Hardware Hash Functions - An Intrinsic Approach, in *Proceedings of the Second International Workshop on Biologically Inspired Approaches to Advanced Information Technology (BioADIT 2006)*, pp. 64-79, 2006.

[17] H. Widiger, S. Kubisch, T. Bahls, and D. Timmermann, A Simplified, Cost-Effective MPLS Labeling Architecture for Access Networks, accepted for publication at the *World Telecommunication Congress* to be held Budapest, April 29th - Mai 3rd, 2006.

[18] `http://www.xilinx.com`

[19] X. Yao and T. Higuchi, Promises and Challenges of Evolvable Hardware. *IEEE Transactions on Systems, Man, and Cybernetics - Part C: Applications and Reviews*, **29**(1):55-78, 1999.