Proceedings of the 2007 IEEE Workshop on
Evolvable and Adaptive Hardware (WEAH 2007)

Accelerating the Evolution of Evolvable Hardware-based Packet Classifiers

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

***Abstract*— In modern networks, the requirements towards network equipment rise together with the bandwidth. Customers and internet service providers ask for more and more services like Voice-over-IP, IP-TV, and data services with a dependable quality. To satisfy these demands, the requirements towards packet classification as key functionality in network equipment, e.g., routers become overwhelming. We developed a packet classifier based on an evolvable hardware hash function and investigated its performance with real world data. The performance did show a reasonable degradation compared to random numbers. The computation time, which is required for evolving one generation in the genetic algorithm, corresponds to the actual fitness. We did find possibilities to maximize the speed of fitness evaluation by taking advantage of the fact, that the whole packet classifier including the fitness evaluation module is a pure hardware implementation based on FPGA technology. We were thus able to increase the performance of the evolvable packet classifier significantly while limiting the additional required hardware resources.**

I. Introduction

Networking equipment hardware, of which routers are the most widely known representatives, is the basis of any digital communication network. This infrastructure controls all the network traffic. Its performance and features determine the services and quality of data communication. A prominent example is the router. A router's main task is to route incoming data packets from input ports to proper output 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. In more complex situations, the router may also consider some of the packet's user data content. Other examples are the functionality of MAC Address Translation (MAT) [1] or MPLS-Labeling [2] [3], which are utilized in Access Networks. These functionalities to route or manipulate data packets base on packet classification. In summary, a module processes every incoming packet, modifies it, and forwards it to a desired port particularly depending on the packet's header fields. This problem is known as the packet classification problem. In addition, increasing numbers of input/output ports and raising bandwidths on each port [4] as well as increasing quality-of-service (QoS) demands request classifiers to process incoming packets as fast as possible. Thus, designers have to construct classifiers with very low latencies, in order to assure traffic and QoS demands, for example, in a Voice-over-IP (VoIP) session.

Figure 1 illustrates a straight-forward approach: A classifier maintains a potentially large data base, which defines how to process incoming packets. A single rule of such a data base can be formulated like:

```
if (frame.src_MAC equals 00:14:22:F5:5A:FF)
then replace-by_00:24:F5:6D:33:0A.
```

In the remainder of this paper, a packet's content that is responsible for selecting a rule is referred to as the *key*.

Conversely, for each incoming packet, the classifier 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 number of rules in the data base, a sequential search through the data base requires an average of $n/2$ memory lookups 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 increase as well. With a data base size of up to 160,000 [5] in state-of-the-art routers, a sequential search might become too expensive in terms of latency and throughput. In other words, the mechanisms for finding the correct routing rule for a given key 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 [6]. Unfortunately, the routing profile of a single router changes over time, which would require adaptive hash tables in order to operate efficiently in time. Accordingly, previous research [7] provides a proof-of-concept that hash tables can be directly employed in hardware.

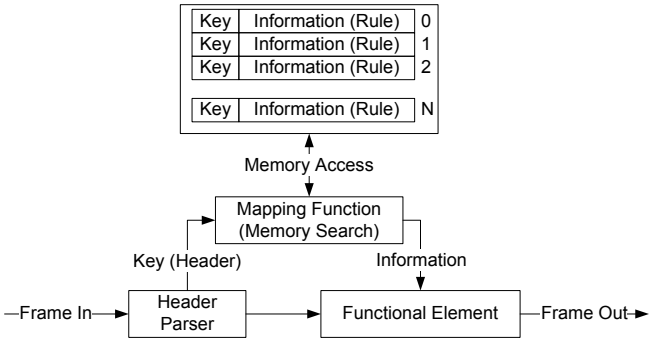

Fig. 1. A packet classifier utilizes a data base to map every incoming packet onto the proper output port.

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. For random numbers as keys in the data base, the developed system did show very promising results [8]. Since this paper investigates the behavior of the system with real world data and possibilities of performance improvements, Section II provides a detailed description of the developed hardware platform. As the hardware platform has been developed and evaluated for random numbers in previous research [7] [8], this paper focuses on how the hardware platform behaves under real world conditions and how to accelerate evolvement by utilizing possibilities, offered by modern FPGA hardware. To this end, Section III describes the experimental setup in full detail. The results, as presented in Section IV, indicate that real world data shows a performance degradation compared to random values and thus deteriorates the speed of evolvement. In order to speed up the evolutionary process, Section V describes the fitness evaluation process in detail and presents different ways for improving the evaluation process. Finally, Section VI concludes with a brief discussion.

II. Overview on System Architecture

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, before the developed hardware classifier is described.

A. Hash Functions: Construction and Properties

In general, a search algorithm of any kind is required when 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 $2{\times}65,536$ 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 $\mathrm{card}(\{h(x)\}) \ll \mathrm{card}(\{x\})$ than the argument domain $\mathrm{card}(\{x\})$. Assume, for example, a packet classifier (router) with 2^{10}=1024 rules and

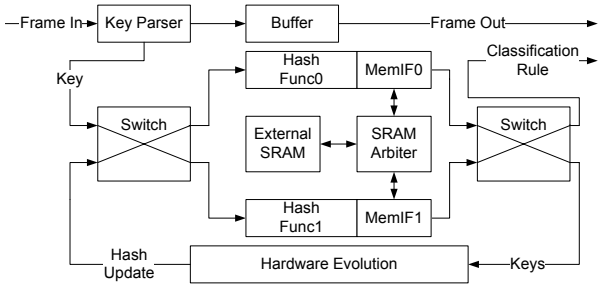

Fig. 2. The implementation of the hash function in hardware.

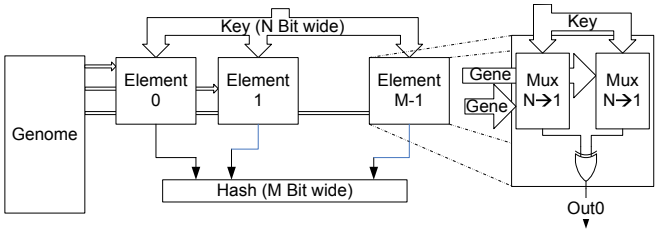

Fig. 3. This hardware classifier platform has been developed in previous research [7]. A genome feeds $\lg n$ elements, which select two bit positions from k-bit-wide keys by a number of multiplexers.

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 collisions 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 collisions. Whether or not the number of collisions 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 [7]. 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 1). 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 2 also features a second hash function, which allows for online updates, and thus, continious 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 collisions, of both. Depending on the actual performance, the platform can utilize either of the two hash functions by properly configuring the two switches. Each of the two hash functions is 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 $s = 2 \cdot 5 \cdot 10 = 100$ in the example discussed above. The Subsection II-C explains how this configuration is done in hardware.

The 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.

C. Realizing Hash Functions in Hardware

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 [7].

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 provide a framework that mainly consists of genetic algorithms [9], evolutionary programming [10] [11], and evolution strategies [12] [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.

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 $p_m = 1/s$. Recombination is not applied, since previous research [8] did show, that recombination has no positive effect in this certain case.

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 a (μ,λ)-GA, this paper also considers the best parent for selection, in order to avoid any deterioration, also known as elitism. In order to achieve a permanent online adaptation, the *hardware* implementation of the genetic algorithm is an integral part of the entire system. Since the goal of the optimization process 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 $k = 32$ bits from a data base of a BGP-Router with 171,000 entries [14]. 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. As described above, the hardware platform has already been realized and used as a proof of concept. This paper uses SystemC simulations, to investigate the classifier's performance. SystemC is a hardware description language (HDL) extension of pure C++. We use a SystemC model in order to remain consistent with the previous research [8]. The simulation runs were performed with a (1+6)-GA. Larger population sizes were not useful as one single run with a $\lambda = 6$ over 1000 generations requires approximately 6 hours of simulation time. It may be mentioned here that the resulting execution time is the only noteable difference between the simulation and the actual hardware platform.

The evolution speed however, is evaluated with the actual FPGA implementation. The limited resources of the development board (Xilinx ML405 with Virtex4-FX20 FPGA [15]) restrict the possible number of keys to 32,768 as well. A (4,12)-GA was implemented to investigate the effects of the improvements for the evolution speed.

IV. System Performace in Real World Environment

With randomly generated keys, the evolvable hardware hash function shows great performance [8]. No matter how many keys were examined, the system never requires more than two memory accesses in average to find the corresponding rule to each key. The number of required accesses is nearly constant between 4,096 and 131,072 keys.

When selecting a (1+6)-GA and 32,768 random 32-bit wide keys, the first generation has a fitness of about 100,000. That means, the 32k keys create 100,000 collisions, when being inserted into a memory. After evolving the genome for 1,000 generations, the fitness value reaches about 12,500, as Figure 4 indicates. That results in less than 0.5 collisions per key or

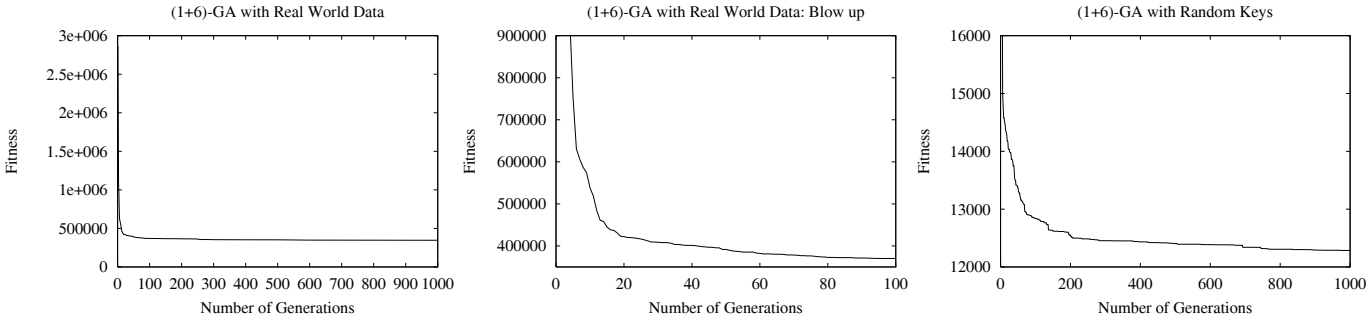

Fig. 4. The Behaviour of a (1+6)-GA with real world data versus 32k randomly generated keys.

1.4 memory accesses on average. These results however, have only limited significance. Data bases of target applications, i.e., router's forwarding data bases (FDBs), are usually not as equally distributed in the search space as random values. The FDB of the BGP-Core router, which we use for the measurements of a real world environment contains 171,000 entries. Each entry consists of a destination IP address (key) and an output port (rule). All keys have a value of `x.x.x.0`. Effectively, a core router uses only the first 24 bits of an address to make its decision. Thus, there are only 24 bit, which contain usable information for packet classification. We thus expect the performance to be worse with that real world data.

With BGP router data the packet classifier does *not* function as well as with random numbers. As Figure 4 clearly shows, already the first fitness value of the (1+6)-GA is 28 times larger than the one for random numbers. 2.8 million collisions are the average initial value over 10 simulation runs. The final fitness value after evolving 1,000 generations is with 350,000 quite a drawback compared to the random numbers as well. However, considering the size of the data base that is being searched, finding each memory entry with 12 memory accesses is still a good value. A more crutial detriment is that the number of required memory accesses is not independent from the size of the database any more. The random numbers always showed approximately 1.4 or less memory accesses per key, no matter if the data base size was 4k or 128k. But with the real world data, the system behaves differently. Figure 5 clarifies this observations. It appears, there is a roughly linear dependency between the size of the data base and the required memory accesses for the higher values. 32k keys required 12, 64k keys 23, and 128k keys 40 memory accesses per key.

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, a smart initialization (SI) process is used that 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. As derives from the research presented in [8], for random data, SI *does* have a positive effect on the hash performance in the earliest stages of the evolutionary process. Although there is no positive effect in latter stages, this behaviour enables the system to function properly and with high performance from the very beginning after the start. SI has a positive influence on real data as well. However, with an increasing number of keys, the positive effect diminishes rapidly. When computing 128k keys, a randomly generated genome creates a slightly fitter first generation than the genome created by SI. It can be concluded, that SI has no advantage over simple random genome initialization for large keysets.

These results lead to the following consequences for further research: First, the overall performance for real world data must be improved. That might be possible by modifying the architecture of the hash function. Second, the evolution process itself must be accelerated. That is because the speed of the fitness evaluation and thus of the whole GA depends on the number of collisions each genome causes. The consequence for real world data are very slow evolution and very poor classification results in early stages of the evolutionary process. Furthermore, adoption processes to changes in the key set are very slow. The following section deals with the problem of accelerating the fitness evaluation.

V. Acceleration of Fitness Evaluation

The fitness of each individual of the offspring is determined by the number of collisions that occur when inserting all keys

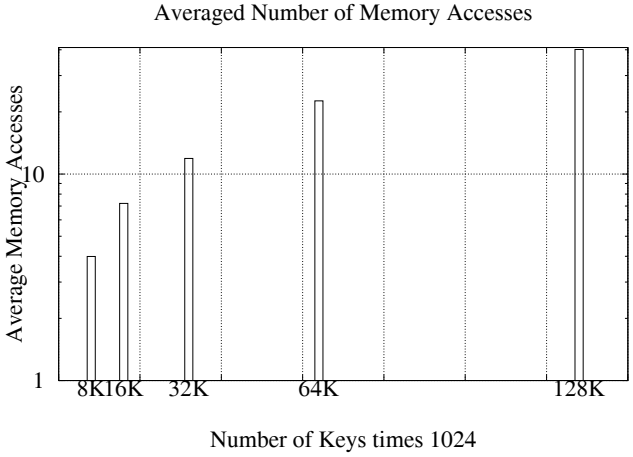

Fig. 5. The Behaviour of a (1+6)-GA with real world data and different key set sizes.

into a memory. The number of collisions, i.e., the fitness is computed by a hardware module. The structure of that module is drawn in Figure 6. There is a one bit wide memory in the module. It consists of as many entries as there are memory positions in the key memories, i.e., two times the number of keys. Each bit denotes, if the memory position is already used by a key or if it is void. Before evaluation of an offspring begins, each entry is reset to '0'. Before evaluating the fitness of one offspring, the actual genome is written to the hash function in order to configure it properly. The evaluation process is achieved by presenting all existent keys one after the other to the fitness evaluation module. Each key is hashed by the hash function with the offspring's actual genome. Then, the memory at address h(key) is read. If it is free, i.e., the entry has the value '0', the value is set to '1'. If the memory is already occupied (the entry is '1'), another key has been inserted at the actual memory position. Thus, a collision counter is incremented, the next memory address is set to h(key)+constant, i.e., the collision resolution is performed, and the memory entry is read again. This way, all keys are inserted into the memory and all collisions that occurred during that process are counted in the collision counter. When all keys are inserted, the collision counter thus specifies the quality of the actual genome. This fitness value together with the actual genome is presented to the next functional module performing the parent selection. The fitness derives directly from the collision counter.

Subsequently, a genome forms a perfect hash function when the collision counter is zero. The time required for evaluating one individual depends on the number of memory accesses required, i.e., on the number of keys k and the number of collisions occurring:

$$T_i = \sum_{j=1}^{k} \tau \cdot (1 + coll_j) \tag{1}$$

The computation time thus decreases from generation to generation, as the fitness improves. To evaluate one generation, the whole offspring has to be computed:

$$T = \sum_{i=1}^{\lambda} T_i + T_{reconf}; \forall j < k : T_j < T_k \tag{2}$$

In case the fitness improves from one generation to another, it is required to rehash the memories for the lookup. This reconfiguration time has to be taken into account for the overall evaluation process. This part of the equation cannot be optimized as it is not a direct part of the fitness evaluation process. It is rather constant.

As Figure 8 shows, the original hardware prototype required roughly 250 seconds to evaluate 200 Generations with a (4,12)-GA. We made ten runs with a random selection of 32768 keys out of the 171000 keys from the data base. As can be expected, the evaluation process speeds up rapidly from generation to generation. The latter generations are much fitter than the earliest ones and the number of collisions thus decreases. The actual configuration of the classifier required 5416 Slices and 18 BRAMS of hardware resources. These performance data refers to the initial system without any optimizations in the fitness evaluation process.

A. Early Termination of Fitness Evaluation

In the standard implementation, the fitness values of all offspring are evaluated completely. After determination of all fitnesses, the next parent generation is selected out of the offspring and the fittest parent. This procedure is unnecessary inefficient. As the selection scheme is static, no further fitness evaluation of an offspring is required, if its fitness is already below the fitnesses of μ other offspring. In that case, the actual offspring cannot be selected for the next generation, even if its fitness value would not rise through further evaluation. Continuing the evaluation process is thus just a waste of time and has to be cancelled. The next individual of the offspring can be instantly evaluated. In the non-optimized case, the time to evaluate all offspring for one generation arises from (2). By early termination (ET), in the best case, the time required for evaluation the whole offspring is given in (3).

$$T = \sum_{i=1}^{\mu} T_i + (\lambda - \mu) \cdot T_\mu + T_{reconf} \tag{3}$$

That is the case, if the μ fittest individuals of the offspring are evaluated first. Of course, if the μ fittest individuals of the offspring are evaluated last, no improvement can be reached by ET. That is, because in the worst case, still every offspring has to be evaluated completely. However, the probability that no improvement is reached in a single generation is very low. The probability, that with μ parents and an offspring of λ is: $\mu!/\lambda!$. Thus, for a (4,12)-GA that probability is $\frac{1}{19958400}$. That means, there is an improvement in nearly any case. This

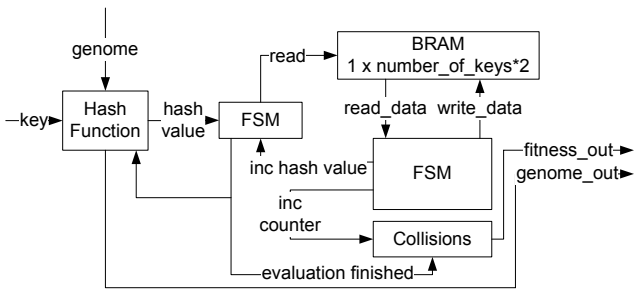

Fig. 6. Architecture of Fitness Evaluation Module

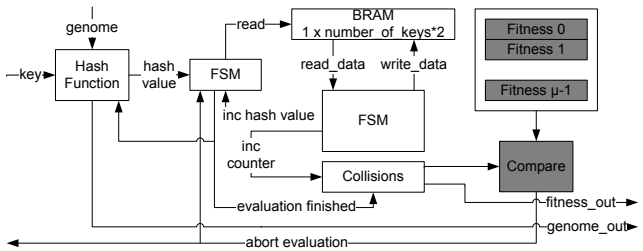

Fig. 7. Architecture of Fitness Evaluation Module with Early Termination

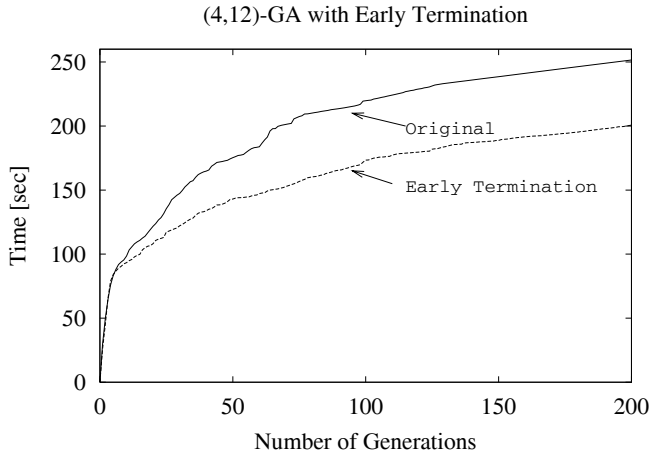

Fig. 8. Time required to evaluate 200 generations with a (4,12)-GA with implemented ET and 32K keys in hardware

optimization can even be expanded. As stated above, the parent selection uses a (μ,λ)-elitist scheme. Meaning the fittest parent is evaluated for the next generation together with the offspring. Thus, the fittest parent is used for determination of ET of the fitness evaluation as well. If there are already μ fitter indiviuals of the new generation including the fittest parent than the actual individual, the fitness evluation has to be stopped. Because there is no chance for being selected for the next generation. Figure 8 illustrates the performance gain, which was accomplished by implementing ET for fitness evaluation. It pictures the degree of improvement over time when evolving the classifier with 32k real world values. The x-axis shows the number of generations evaluated. The y-axis shows the real computation time required for evolving 200 generations. While the classifier with the original configuration required averaged 250 seconds, the performance with ET improved the speed by 25% to 200 seconds. That can be considered a substantial gain!

To implement ET into the evolution process, only minor changes to the fitness evaluation module are required (Figure 7). In addition to the standard logic, a register set need to be implemented. It stores the fitness values of the μ fittest individuals of the offspring and the fittest parent. The collision

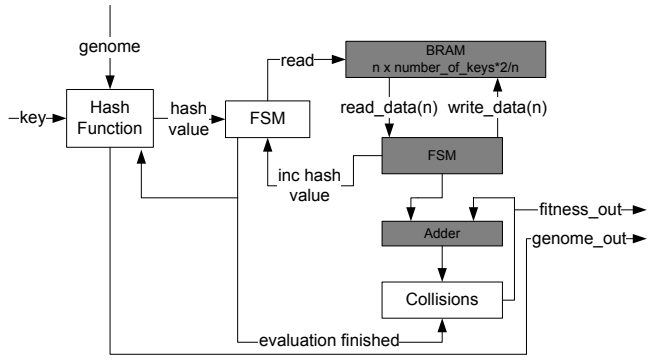

Fig. 9. Fitness Evaluation Module with Memory Interleaving

counter is compared against the worst of those fittest individuals. When the fitness gets worse, the termination of the actual evaluation is signalized and the evaluation stops immediately. The classifier with implemented evaluation termination required 5592 slices of hardware resources. Compared to the 5416 slices of the original system, the size of the system increases by only 3%. With a speed-up of about 20%, the additionally required hardware resources are negligible.

B. Memory Interleaving

Another way of utilizing the advantages of a hardware implementation is checking for many collisions in parallel. In the ordinary system a memory position is checked to determine, if it is already used or if it is void. In case it is used, the memory is searched until an empty memory position is found to insert the actual entry. This process can (partially) be parallelized. When checking not only the memory position at h(key) but, for example, the next three possible entries as well, lots of memory accesses can be saved. The evaluation time is reduced from

$$T_i = \sum_{j=1}^{k} \tau \cdot (1 + coll_j) \tag{4}$$

to

$$T_i^n = \sum_{j=1}^{k} \tau \cdot \left(1 + \left\lfloor \frac{coll_j}{n} \right\rfloor\right) \tag{5}$$

with n denoting the degree of the memory interleaving. This improvement can be reached with virtually no additional hardware costs. The required Block RAM needs only to be reconfigured from $1 \times number\_of\_keys \cdot 2$ to $n \times \frac{number\_of\_keys \cdot 2}{n}$. The additional logic resources are a 32-bit adder and an n-bit comparator. In the implementation, the memory interleaving (MI) extensions require maximal 3% of extra logic. On the other hand, the acceleration of evolution is impressive compared to the expense. As emanates from Figure 10, the use of 2-bit MI speeds up the process by nearly 40%. The use of a 4-bit MI reduces the computation time from 250 seconds to even 108 seconds. We did not expand the memory interleaving to 8-bit MI. Even though, we do not expect a substantial rise in the hardware requirements, the minimal clock rate of the hardware implementation of the classifier was not reached with eight-bit interleaving. However, in next generaration FPGAs or with higher speed grades, the use of 8-bit or even 16-bit MI is not an issue.

C. Parallel Fitness Evaluation

As described above, the fitness evaluation computes all offspring serially in the original system. However, the fitness of every offspring can be evaluated independently from the other individuals. Being a hardware implementation enables parallel fitness evaluation (PFE) for the offspring. Thus, the evaluation process for every generation can be accelerated significantly. The time an evolution process requires is given in (2). It represents the sum of all evaluation times the individuals of the offspring require. In the extreme case, PFE can be

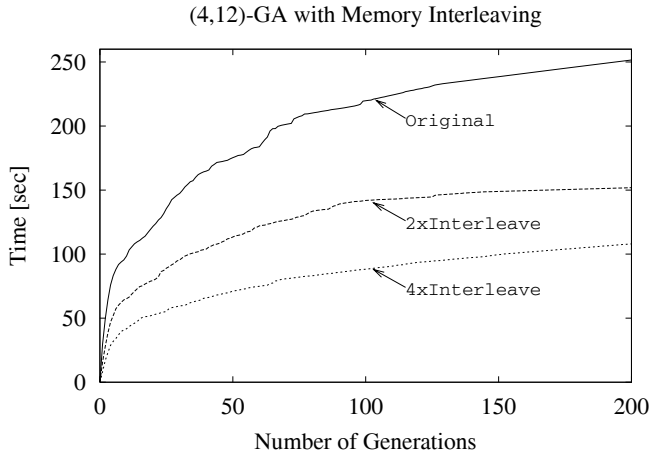

Fig. 10. Time required to evaluate 200 generations with a (4,12)-GA with implemented MI and 32K keys in hardware

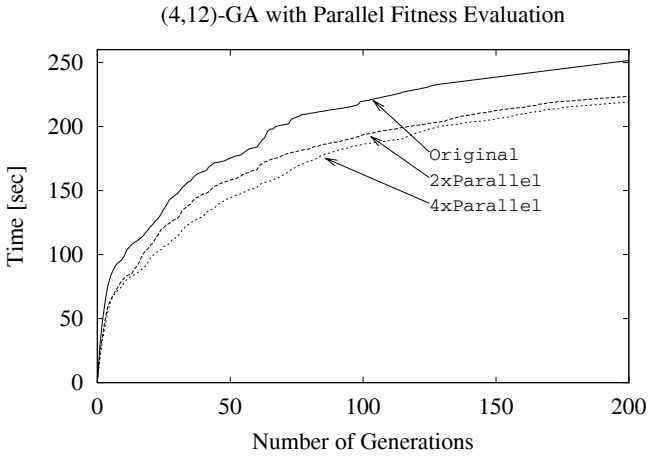

Fig. 12. Time required to evaluate 200 generations with a (4,12)-GA with implemented PFE and 32K keys in hardware

applied to all individuals of the offspring. That would require λ hardware modules for fitness evaluation. Each one computes the fitness of one individual. The size of the hardware system rises by 10 and 32% when using $2\times$ or $4\times$ PFE. However, the overall computation time cannot decrease from T to T/λ as could be expected. There are two reasons for this: First, the time for reconfiguring the memories and for rehashing T_{reconf} is constant and cannot be reduced. Second, the fitness evaluation is based on virtually hashing all keys. After on key is evaluated, the next one enters the functional module and so on. The evaluation process works as follows: The key is transferred to all evaluation modules. It is hashed to each individual memory. Collisions are resolved. Finally the memory entry is done. This task has to be performed by every PFE module individually. Only when all individuals have hashed the actual key successfully, the next key can be read. Thus, the time required for evaluating a single key depends on the individual generating most collisions (6).

$$\sum_{i=1}^{k} \overset{\lambda}{\max_{j=1}} (T_{i,j}) + T_{reconf} \tag{6}$$

We can expect that the degree of improvement aligns to T/λ in the process of evolution. How far depends on the fitness of each single individual of the offspring (7).

$$\lim_{coll \to 0} \overset{\lambda}{\max_{j=1}} (T_{i,j}) = \overset{\lambda}{\min_{j=1}} (T_{i,j}) \tag{7}$$

How the increase of speed through parallel fitness evaluation behaves in the simulations is presented in Figure 12. For both $2\times$ and $4\times$ PFE, an improvement can be determined. However, with 12 and 15% respectively, the speed gain compared to the resource investment is surprisingly low. The drawbacks, which were discussed above, have obviously a very high influence when the individuals of the offspring cause a relatively high number of collisions. Consequently, one must state that compared to the costs in hardware the speed gain does not suffice. That is at least, if no combination with MI is utilized.

D. Combination of all Improvements

All three aforementioned methods for acceleration of the evolution speed (ET, MI, PFE) are independent of each other. It is thus unproblematic to combine all approaches. When combining MI and PFE the two approaches can be expected to influence each other positively. The positive influence of ET on the evolution process on the other hand is reduced when combined with PFE. In the extreme case, when λ parallel modules are used, there cannot be any gain at all caused by ET. Utilizing MI causes the gain by PFE to be better than PFE vs. the original implementation. That can be clarified

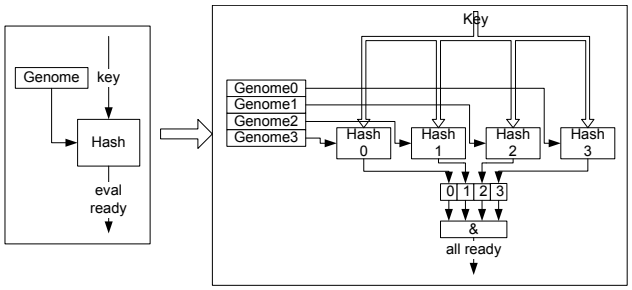

Fig. 11. Parallelization of Fitness Evaluation

Module	Slices	Increase [%]	BRAMs	Evolution Time [sec]	Speed Gain [%]
Original	5416	-	18	251	-
ET	5592	3.25	18	200	25
2x PFE	5945	9.77	22	223	12
4x PFE	7203	32.99	30	219	15
2x MI	5531	2.12	18	152	67
4x MI	5572	2.88	18	108	132
Combination	7473	37.98	30	85	230

TABLE I

RESOURCE CONSUMPTION FOR DIFFERENT IMPLEMENTATION VARIATIONS

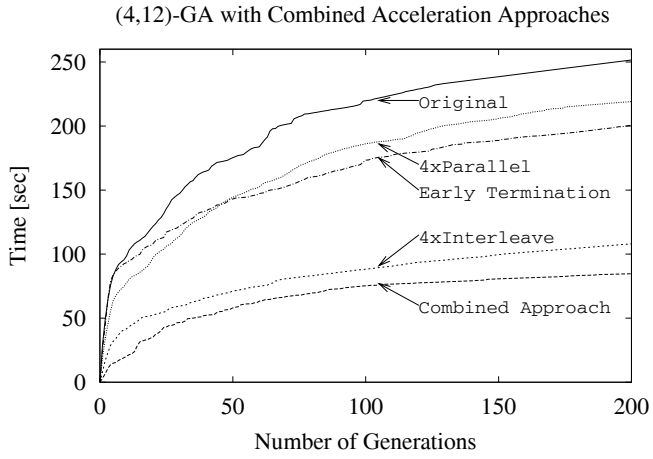

Fig. 13. Time required to evaluate 200 generations with a (4,12)-GA and 32K keys in hardware. The best performance is shown by the combined aproach including $4\times$ MI, $4\times$ PFE, and ET

with an example. Considering two genomes create with the same key 1 and 7 collisions: In the original implementation $2 + 8$ ($1 + 1$ insertions and $1 + 7$ for collisions) sequential memory accesses are required to resolve the collisions. Using a PFE scheme with two modules leads to 8 sequential memory accesses. That is because the 2 accesses of the first genome are executed in parallel to the 8 accesses of the second genome. However, the first hardware module has to wait until the second one has computed the key. Implementing just $4\times$ MI reduces the accesses to $1 + 2$. But combining MI and PFE reduces the sequential memory accesses to just 2. Consequently, MI and PFE together reduce the idle-time of the parallel modules significantly, and thus increase the evaluation speed. A fitness evaluation with $4\times$ MI, $4\times$ PFE, and ET was implemented on the target platform. The hardware costs amount to 7473 slices and 30 BRAMS, which means an increase of about 38 and 67% respectively. As shown in Figure 13, the necessary computation time on the other hand reduces from 251 to 85 seconds. Thus, a total speed-up of 230% is finally accomplished!

VI. Conclusion

In order to address the problem of fast packet classification in state-of-the-art network routers, this paper has applied real world router data from a BGP-router on an evolvable packet classifier. It turned out that compared to randomly generated keys the performance of such a system decreases. However, on a 32k large data base and with 12 memory accesses per key it is still reasonably fast.

Since it allows for faster adaptation to changing rule data bases, the evolution process was accelerated. To achieve this goal, different acceleration methods were developed, implemented in hardware, and investigated on a development board. All three of them (ET, MI, and PFE) turned out to have positive effects on the speed of the evolution process and acceptable additional costs in terms of hardware resources. The combination of all methods increased the evolution process by 230% in total. The costs increased by only 38% for slices of logic and 67% for Block RAMs of the selected FPGA.

The packet classifier with the improved fitness evaluation algorithm has been implemented in hardware using the VHDL description language. In a Xilinx Virtex4-FX20 FPGA [15], the system consumes 7473 slices of logic and runs at a clock speed of 120 MHz. At this speed, the classifier is capable of performing more than 10 million classifications per second, when assuming 12 memory accesses per classification (as indicated by the simulation results).

Further research will be dedicated to an analysis, if changes in the architecture of the hardware hash function can improve the quality of the packet classification of real world data significantly. It will also be investigated if an adaptive mutation rate has a positive effect on the results and/or the speed of the evolution process. Further research will also be dedicated to the effects of changing the fitness function towards the consideration of the frequentness of different keys in the data path of the classifier. That is because it would be better if commonly occurring keys would be looked up quicker than seldom occurring keys. That traffic profiling is supposed to have positive effects on the system's throughput, even if the actual fitness function would not indicate it.

References

[1] S. Kubisch, H. Widiger, T. Bahls, and D. Timmermann, Wirespeed MAC Address Translation and Traffic Management in Access Networks, in Proceedings of World Telecommunications Congress (WTC 2006) on CD-ROM, *World Telecommunication Congress* Budapest, Hungary, 2006.

[2] H. Widiger, S. Kubisch, T. Bahls, and D. Timmermann, A Simplified, Cost-Effective MPLS Labeling Architecture for Access Networks, in Proceedings of World Telecommunications Congress (WTC 2006) on CD-ROM, *World Telecommunication Congress* Budapest, Hungary, 2006.

[3] H. Widiger, S. Kubisch, T. Bahls, and D. Timmermann, A Hardware Solution for MAT, MPLS-UNI, and TM in Access Networks, *Proceedings of the 31st Annual IEEE Conference on Local Computer Networks (LCN),* pp. 272-279, 2006.

[4] J. Cioffi, M. Mohseni, B. Lee, V. Pourahmad, and M. Brady, MIMO Spectrum Management, *All Star Network Access Workshop* Geneva, Switzerland, 2004.

[5] 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.

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

[7] 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.

[8] S. Salomon, H. Widiger, A. Tockhorn, D. Timmermann, Rapid Evolution of Time-Efficient Packet Classifiers, *Proceedings of the IEEE World Congress on Computational Intelligence (WCCI 2006),* pp. 2793-2799, 2006.

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

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

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

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

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

[14] `http://archive.routeviews.org/oix-route-views/`

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