Packet Classification with Evolvable Hardware Hash Functions – An Intrinsic Approach

Harald Widiger, Ralf Salomon, and Dirk Timmermann

University of Rostock,
Institute of Applied Microelectronics and Computer Engineering,
Richard-Wagner Str. 31, 18119 Rostock-Warnemuende, Germany
{harald.widiger, ralf.salomon, dirk.timmermann}@uni-rostock.de

Abstract. Bandwidth demands of communication networks are rising permanently. Thus, the requirements to modern routers regarding packet classification are rising accordingly. Conventional algorithms for packet classification use either a huge amount of memory or have high computational demands to perform the task. Using a hash function in order to classify packets is promising regarding both memory and computation time. However, such a hash function needs to be of high performance and cheap in hardware costs. These two design goals are contradictory. To limit the costs of a hardware implementation, known good hash functions, as used for software implementations of encryption algorithms, are applicable to only a limited extend. To achieve the goals mentioned above, an adaptive hash function is needed. In this paper, an approach for a hardware packet classifier using an evolvable hash function is presented. It consists of an evolutionary algorithm which is entirely implemented in hardware.

1 Introduction

In state of the art communication technology, an increasing amount of data has to be transferred. The bandwidth demands of communication networks are rising permanently. Not only the bandwidth demands but also the service demands on state of the art network equipment rise as well. Voice over IP (VoIP) traffic, for example, requires very low latencies. The diversification of data streams in routers driven by the raising quality-of-service (QoS) demands of customers and internet service providers accelerates the packet classification problem in routers rapidly.

1.1 Packet Classification Problem

Network routers must offer a huge variety of services on different flows. These services comprise routing, rate limiting, access control to networks, virtual bandwidth allocation, traffic shaping and policing, and service differentiation. In order to distinguish between different flows, nearly all network components both at the edge and in the core of a network need a packet classifier. Many of the

A.J. Ijspeert et al. (Eds.): BioADIT 2006, LNCS 3853, pp. 64–79, 2006.

aforementioned services are time sensitive. That is why network routers need to classify the packets at wire speed in order to add as less latency as possible before making service decisions.

The classification of packets is based on rules. All rules are stored in a database. The packet classification problem is to determine the rule, that matches for an incoming packet. The database has to be searched in order to find the matching rule. The search must be performed as fast as possible. With higher bandwidths and a large number of rules, a huge database has to be searched in a shorter time. Thus, packet classification is a severe problem in state of the art communication technology for which only limited hardware resources are available. IP lookups in routers for example should be as fast as possible. Conventional algorithms [1] must make a tradeoff between classification speed and memory demands. The algorithms are either implemented in software or in hardware. The software implementations usually lead to comparative low memory requirements but very high search latencies. Many of them use tree structures to classify packets. The HiCuts algorithm [2], for example, partitions the search space guided by heuristics. Each search leads to a leaf, which consists of a small number of rules. The leaves can then be searched linearly to find the best match. The hardware based algorithms on the other hand can perform in wire speed. But therefore either a huge memory amount or specialized and thus expensive hardware or memories like ternary content addressable memories (TCAMs) are required. A TCAM memory array stores all rules (N) in decreasing priority. An input key is compared to all rules in the array in parallel. The N-bit vector indicating all matching rules is read by an N-Bit priority encoder, which indicates the address of the highest priority match. The address is used to index a random access memory (RAM) to find the action associated with the prefix. Besides the potential high costs, another drawback of hardware implementations is the very limited number of rules that can be stored [1].

A solution of the packet classification problem can be the use of hash functions. By the use of a hash function, the two main demands to a packet classifier can be met. Hash functions have a search complexity of ideally $O(1)$. Thus, they are independent of the number of elements searched. With $O(N)$, memory requirements scale only linearly with the number of classification rules. This paper presents a hardware packet classifier that is based on a hash function. Section 2 gives a short overview over the basics of evolutionary algorithms. In section 3, hash functions in general and the evolvable hash functions used for the packet classifier in particular are characterized. Section 4 summarizes the simulation results of the implemented hash functions. Section 5 details the implemented hardware architecture of the packet classifier. Section 6 concludes this paper and presents an outlook to future work.

2 Evolutionary Algorithms

Evolutionary or genetic algorithms are search algorithms based on the mechanics of natural selection and natural genetics [3]. They combine survival of the

fittest among string structures with a structured yet randomized information exchange to form a search algorithm with some of the innovative flair of human search. In every generation, a new set of artificial creatures based on the bits and pieces of the old generation is created. Being randomized, genetic algorithms are not simply random walks trough the search space. They efficiently use historical information to speculate on new search points with expected improved performance.

Genetic algorithms work with a coding of a parameter set, instead of the parameter set itself. The search is not done from a single point but a population of points. For determining the quality, a fitness function is used. This fitness function measures the quality of an artificial creature regarding its purpose. In contrary to traditional methods, genetic algorithms use probabilistic transition rules rather than deterministic ones. The mechanism of a simple genetic algorithm involves nothing more than copying strings and swapping partial strings, bit vectors respectively. These transitions are called operators. A simple genetic algorithm is composed of three operators: reproduction, crossover, and mutation.

Reproduction is a process in which individuals are copied depending on the fitness function. A limited number of individuals (the fittest ones) is copied to form the base of the next generation.

Crossover is a process that simulates sexual reproduction (Figure 1). Parts of the bit vectors of two individuals A1 and A2 (the parents) are exchanged producing two offspring (A'1 and A'2) having features of both parents. The individuals created by the reproduction form a mating pool. Members of the pool are mated at random. An integer value k between 1 and the length of the bit vector minus one (l-1) is drawn uniformly. Two new bit vectors are created by swapping all bits between k+1 and l.

Mutation is a process in which bits of the bit vector of the individuals are inverted at random positions. The mutation probability is relatively low. A mean mutation frequency of one mutation per number of bits in the bit vector obtains good results [3]. However, the best mutation rate may be different for each application. If the rate is too high, a random search is performed rather than a genetic algorithm. If it is too low, the speed of the quality improvement of the genetic algorithm is limited needlessly.

If the genetic algorithm is implemented in a hardware structure, it is called an evolvable hardware (EHW). According to [4], EHW can be classified into two categories, extrinsic and intrinsic evolvable hardware. In extrinsic EHW the genetic algorithm is performed externally in software. As a result, only the best configuration obtained is downloaded into hardware. This is done once in each

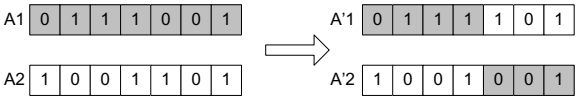

Fig. 1. Crossover Operator

generation. In the intrinsic approach, the hardware itself simulates the genetic algorithm. This has two main advantages. On the one hand, the genetic algorithm is performed much faster in a specialized hardware than it could be in software on a general purpose processor. On the other hand, such an evolvable hardware can operate autonomously in an Field Programmable Gate Array (FPGA) without an interface to a processor with a software system.

3 Hash Functions

As stated in the introduction, packet classifiers have two main demands, high classification speed with low latency and low hardware (memory) costs. By the use of a hash function the two main demands to a packet classifier can be met. Hash functions have a search complexity of ideally $O(1)$. Thus, they are independent of the number of elements searched in. With $O(N)$ the memory need scales only linearly with the number of classification rules. However, it is problematic to find a sufficient hash function. It has to be both high performance and of low hardware costs. It might be easy to find a good hash function for a specific amount of elements out of a huge search space. But because of changing key sets in packet classifiers, a hash function, that used to be sufficient, might be insufficient for a modified key set. A solution for this problem is a permanently adapting and improving hash function. This goal can be obtained by evolutionary computing completely done in hardware. Such an evolvable hardware hash function is proposed here.

Hash Functions map a value X to its hash value h(X). Usually hash functions do a conversion from a large domain to a much smaller domain. In case of a packet classification, i.e., a 32-bit key (destination IP address) is hashed to a 10-bit wide memory address to store rules for 1024 different keys. Thus, the goal is to map 2^m elements from a search space of 2^n to a much smaller search space of 2^m. The quality of the hash function can be determined by measuring the number of collisions, that occur when hashing all keys into memory. Ideally, every element hashes to a different value. In that case, a hash function is perfect. That would mean for the packet classification that the search for a rule corresponding to a key would be done with just one memory access. Finding such a perfect hash function is very difficult. Depending on the algorithm or the hardware structure of that function, there might be no perfect hash function. Normally a hash function is not perfect. Thus, collisions occur when hashing a number of keys. A collision occurs if two different values are hashed to the same value (equation 1).

$$X \neq Y; h(X) = h(Y) \tag{1}$$

Those collisions must be resolved. This can be done in two different ways [5]. One way is to rehash h(Y) with another hash function that hashes an m-bit value to another m-bit value until no collision occurs and a free entry in the hash memory is found. Another way is to perform a linear collision resolution. For a linear collision resolution a constant value is added to the hash value. This is done until a free memory entry was found. The constant must be a prime

number or at least a number which is relatively prime to the number of memory entries. This is required to assure that all existing memory entries are searched before reaching the original entry. In the simplest case the constant is 1. As mentioned above, the quality of a hash function can be measured by counting the number off collisions that occur when all keys are hashed into memory. A perfect hash function would not create any collision. The worst hash function on the other hand would be one that hashes all values to the same hash value. In that case, the maximum number of collisions that would occour is $\frac{n^2-n}{2}$. To limit the number of collisions and to therefore increase the lookup performance, the memory load is usually limited to $\frac{1}{2}$. This means 2^m elements are hashed to (m+1)-bit wide hash values and stored in a memory with 2^{m+1} entries.

Memories accessed by hash functions have the great advantage of fast updates. Both insertions and deletions can be achieved very fast. They have the same complexity as searching. To insert a new entry in a hash memory, the key has to be hashed. Then the memory has to be searched on the bases of $h(key)$ until an empty entry was found. The new entry can be inserted at this position. Deletion is a little more complicated, as the corresponding memory entry cannot simply be freed but must be marked as deleted. Freeing is only possible when rehashing the complete memory. Thus, insertions are not only done when a free memory position is found but also at memory positions that are marked as deleted.

3.1 Evolvable Hardware Hash Functions

To be able to perform many key lookups in a packet classifier with the utilization of a hash function, the hash function should be implemented in hardware. The database of a packet classifier is not static: Permanently rules are added or removed. Thus, a hash function, that used to be sufficient and of good quality for a specific database, gets insufficient with the changes of the database.

A hash function is needed that can be implemented easily and efficiently in hardware. To adapt at any time to an actual set of keys, the hash function shall evolve autonomously. Thus, a complete hardware evolution comes to pass. This is realized by constantly traversing an evolution pipeline comparable with the one in [6]. The system is implemented both as a SystemC software model and a fully synthesizable VHDL description for implementation into an FPGA. A linear collision resolution for the hash functions is used. In the following sections, different hardware architectures of hash functions are explored to determine their potential.

3.2 Hash Architecture 1 and 2

A promising architecture which is high performing and relatively cheap in hardware costs is shown in Figure 2. In the following it is referred to as hash1. It consists of a number of multiplexer elements. The multiplexers are controlled by registers. Those registers form the genome of the hash function. For every output signal, two multiplexer outputs are connected via an xor function. To hash an N-bit value to an M-bit value, $2 \cdot M$ N-to-1 multiplexers are needed. As

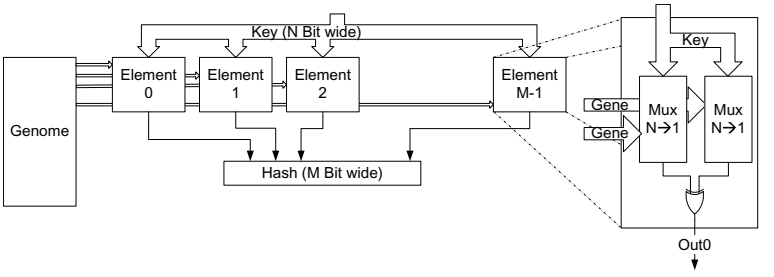

Fig. 2. Architecture of an Evolvable Hash Function (hash1)

every multiplexer can multiplex any of the input bits to its output, controlling a multiplexer demands $\log_2(N)$ bits. These bits controlling the function of the multiplexers form the genome of the hash function. Thus, to hash N bits to M bits, a genome size as stated in expression 2 is needed. To hash 1024 32-bit keys a hash function with a genome size of 100 bit is required.

$$2 \cdot M \cdot \log_2(N) \tag{2}$$

A variant of the above introduced architecture was developed as well (Figure 3). In the following, it is referred to as hash2. Here the genome is enlarged to increase the possibilities for evolutionary development. It is a two-stage architecture. We use the elements consisting of two multiplexers connected by an xor in the first stage. In the second stage, first stage results are mixed up by multiplexers. To hash N bit to M bit a genome size as stated in expression 3 is needed. To hash 1024 32-bit keys a hash function with a genome size of 370 bit is needed.

$$(2 \cdot N + M) \cdot \log_2(N) \tag{3}$$

3.3 Hash Architecture 3

In [7], a hardware architecture of a hash function is presented. In the following it is referred to as hash3. The presented architecture was adapted slightly to improve its performance and to limit the hardware costs. The hash function

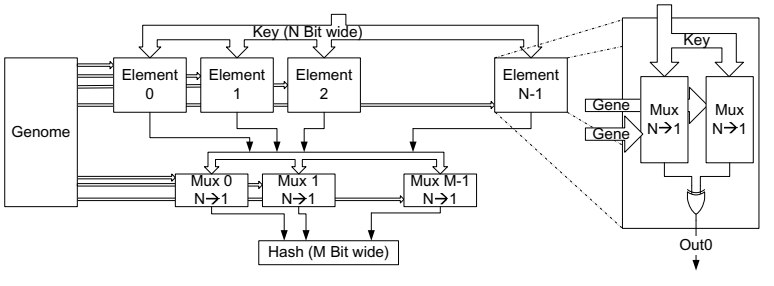

Fig. 3. Architecture of an Evolvable Hash Function (hash2)

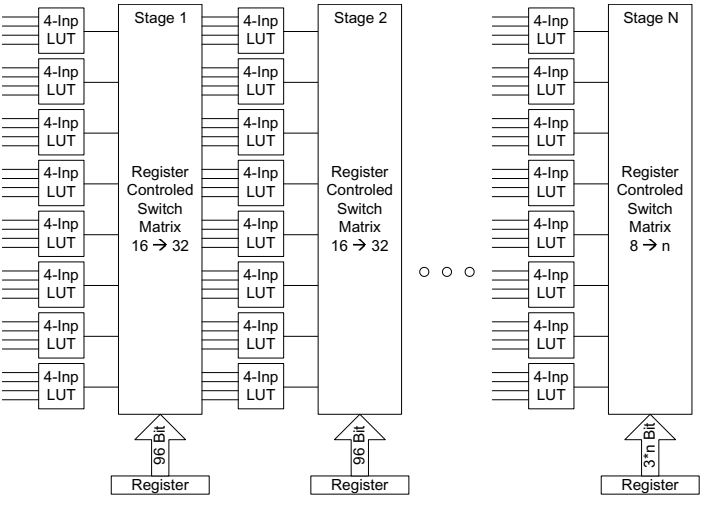

Fig. 4. Architecture of an Evolvable Hash Function (hash3)

matches the structure of FPGAs very well. It mainly consists of four-input-look up tables (LUTs). In those LUTs any logical 4-input function can be performed. To code the function 16 bit are needed. The logical functions are mapped to the FPGAs LUTs. Every slice of i.e. a Virtex2 FPGA comprises of two LUTs. As all LUTs have four inputs, to map an N bit wide key, $\frac{N}{4}$ parallel LUTs are needed for one complete stage. Every stage is followed by a register controlled switch matrix. All but the last switch $\frac{N}{4}$ inputs to N outputs. The last matrix switches $\frac{N}{4}$ inputs to M outputs.

$$\left[\left(N \cdot \log_2\left(\frac{N}{4}\right)\right) + \frac{N}{4} \cdot 16\right] \cdot (S-1) + M \cdot \log_2\left(\frac{N}{4}\right) + \frac{N}{4} \cdot 16 \qquad (4)$$

For the whole function with an N-bit key and M-output bits and a depth of S stages a genome as given in expression 4 is needed. To hash 1024 32-bit keys a four staged hash function with a genome size of 830 bit is needed.

4 Simulation Results

The evolvable system comprises the complete evolutionary algorithm. It works autonomously and without any control of a software system. Thus, it is an intrinsic EHW. It was implemented as a SystemC model. The model is functional identical to the VHDL implementation, which is described in chapter 5. All three hash functions were implemented and evaluated with different key sets of up to hundred thousand 32-bit keys. The keys were generated randomly. The memory load of the hash memory was set to different levels ranging from 37% to 87%. All hash functions evolved over thousand generations.

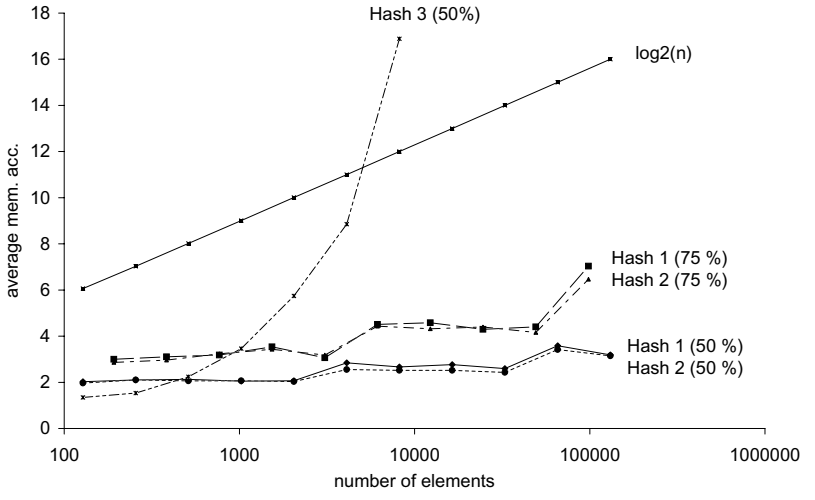

Fig. 5. Lookup Performance of Different Hash Architectures. Average memory accesses needed for finding an entry in the hash memory.

In Figure 5, the graphs reveal, that the architectures of the hash functions performed differently. The hash functions build of multiplexers (hash1 and hash2) showed great performance. The average of required memory accesses over all keys scales only very slightly. For a memory load of 50%, both architectures hash1 and hash2 show a comparable performance. The average number of memory accesses over all keys stored in the memory remains below four even with 2^{17} keys. Finding an entry in a sorted list of 2^{17} keys, sixteen accesses are needed on average. Four accesses outperform the sixteen memory accesses of a sorted list remarkably. The architecture consisting of LUT rows (hash3) showed poorer performance. It scales almost linearly with the size of the key sets. Even with just 8192 keys, an average of 17 memory accesses is needed. Even at this relatively small number of keys, the sorted list outperforms the hash function with just twelve memory accesses. Thus, the architecture is not applicable for implementation in a packet classifier.

When using a memory with a load of 75%, hash1 and hash2 perform well, too. Searching for one of 2^{17} keys requiress six memory accesses on average. Nevertheless the performance has decreased by about 50%. Here we make a tradeoff between search time and memory demands. However, it has to be mentioned that the theoretical upper bound for searching in the memory with the help of a hash function is still $O(N)$. Even if the average memory accesses needed are very low, the worst keys require far more than $\log_2(N)$ memory accesses. This is the case at least for the memories with a load of 75%, as emanating from Figure 6.

It shows that the number of needed memory accesses for particular keys are extremely high. Especially when the memory load is at 75%, the maximum needed memory accesses are at 80 for hash1 and at 166 for hash2 respectively. When the memory load was as high as 87%, the worst case memory accesses

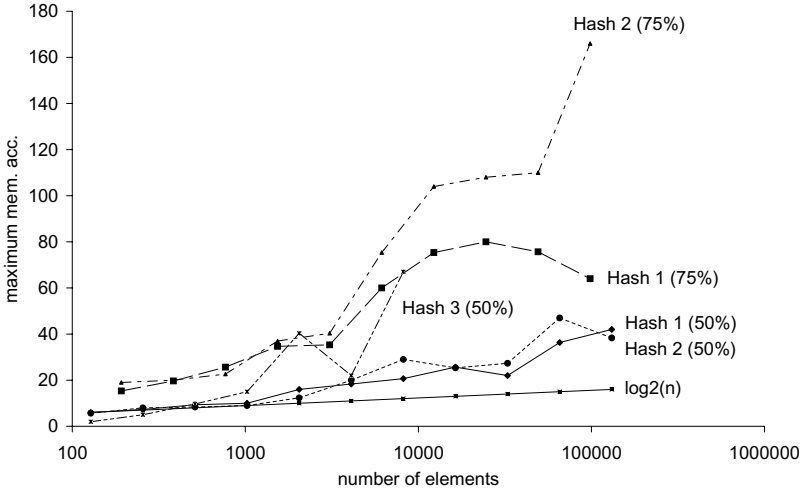

Fig. 6. Lookup Performance of Different Hash Architectures. Maximum memory accesses needed for finding an entry in the hash memory.

needed where at 262 (hash1) and 283 (hash2). Whereas the hash functions with a memory load of 50% show a better performance. Here the maximum needed memory accesses are at 42 (hash1) and 47 (hash2). That is why a memory load of 50% should not be exceeded in a system implementation, as the penalty for the worst case is rising quickly. The memory demand of such architecture has a complexity of $O(N)$. As the hash function evolves constantly, repetitive rehashing of the keys in the memory is required. To do so, there have to be two memory blocks of which one is used in the data path while the other one is rehashed. That means if the memory load is limited to 50%, exactly $4 \cdot N$ memory locations are required to store keys and classification rules.

5 Hardware Architecture of the Packet Classifier

The System was implemented in VHDL. Functionally it equals the SystemC model used for the simulations exactly. The packet classifier consists of two main elements. The data path and the evolution pipeline. The classifier is completely described in VHDL and was implemented into a Xilinx Virtex2 FPGA. The packet classification is done at wire speed. So no external memory is designated to buffer the data packets. Only a FIFO build of internal block RAMs of the FPGA stores the packets until the classification rule is extracted from the memory.

5.1 Data Path

In the data path, incoming packets are parsed and the key is extracted. The packet is stored in a buffer until the corresponding classification rule has been extracted from the memory. Based on the key the packet is classified. The key

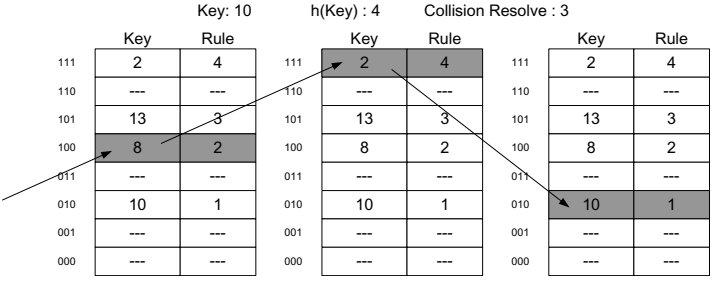

Fig. 7. The hash value for the key 10 is 4. Searching for key 10 leads to memory address 4 in the first run. As the stored key is not equal to 10, the next address is h(key)+3 = 7. As the stored key is not equal to 10 either, the next address is h(key)+3+3=2. At memory entry 2 the right correct is found and the corresponding rule is returned. Three memory accesses were needed.

is hashed by the evolvable hash function. Using the hash value $h(key)$ as start address, the memory is searched. If the stored key does not comply with the search key, a collision occurred. Using linear collision resolution, the next read address in the memory is set to: $h(key) + prime$. The prime number is configurable. Adding the prime number to the read address is repeated until the correct entry or an empty memory entry is found. In Figure 7, an example of such memory access is given. When the correct entry is validated by comparing the search key with the one stored in the memory, the classification rule together with the packet is sent to the output of the classifier. To solve the problem of numerous memory accesses, a small cache can be implemented to the memory module. The eight worst keys, which produce many memory accesses can be held in that cache and therefore be read out in just one memory access. This would improve the overall quality of the packet classifier. However, this feature has yet to be implemented to the packet classifier. The classifiers key parser module extracts the search keys from the incoming packets. The module can extract any combination of bits from a data packet. The bits, the key consists of, can be configured at any time. The bit mask for the key is stored in a memory block which is accessed through a configuration port. The generic architecture of the key parser allows the configuration of the width of the search key at implementation time while the actual bits of the key can be changed while the classifier is in use. In that way, the code guarantees high flexibility.

As the hash function changes during operation of the packet classifier permanently, it needs a repeating reconfiguration. In addition a permanent rehashing of the memory is required. This would interrupt the packet classification process very often for a quite substantial time. This is the reason why the data path consists of two independent hash functions and hash memories. While one path is used for the normal operation of the packet classifier, the other one can adapt to a new evolved and better performing hash function. The reconfiguration of the unused hash function is done without affecting the one used in the data path. Furthermore, the time intensive rehashing of the memory can be done as

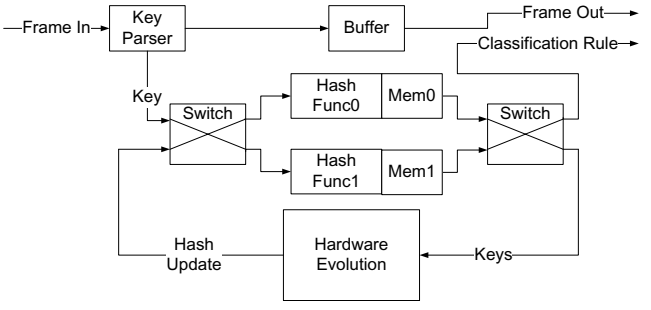

Fig. 8. Architecture of the Data Path

well without interference. The time for the rehashing process depends on the one hand on the number of memory elements to be rehashed and on the other hand on the traffic the classifier is exposed to. To be able to rehash the unused memory, the one utilized in the data path is needed to provide the information that is to be rehashed. In order to not interfere with the functionality, data reads for rehashing can only be gathered when the utilized memory is in an idle state. Memory accesses from the key parser always have the highest priority. Therefore the duration of the rehash process is not determined. When the inactive memory is rehashed and the hash function is reconfigured, the data path is switched to the new configured path. If the maximum memory load is limited to 50% as it is mentioned above, the memory demand of the classifier is 4 times the number of keys. The memory can be implemented either with the internal block RAMs of the FPGA or by using an external memory. A V4FX40 FPGA, i.e., has a total of over 2.5 million bits of block RAM. Assuming a key size of 32 bit and a rule size of 16 bit for classification, 13500 different rules can be stored internally. Therefore, classifiers with small and medium rule sets can be implemented without using external memories.

5.2 Evolution Module

The evolution module performs the whole evolutionary process completely in hardware. It consists of six functional elements. The evolution is performed permanently, stopping only if a perfect hash function was found. Perfect means that all key are hashed to a different value. The evolutionary algorithm used in the hardware is the following: On reset of the system, four individuals, representing four different genomes of the hash function, are generated (pseudo)randomly by a linear feedback shift register (LFSR). The number of bits k, the genome consists of, arises from expressions three, four, or five depending on the chosen hash function. From these four parents an offspring of twelve is generated by random as well. The best of the parents does not have a bias to be chosen with a higher probability. The offspring is then mutated. Due to a simple and efficient hardware implementation the mutation probability p is always as stated in equation 5.

$$p = \frac{1}{2^k}; \quad 2^k \leq l < 2^{k+1} \tag{5}$$

On average, between one and two bits of a genome are mutated. After mutation, the fitness of the offspring f(x) is evaluated. The fitness of an individual is measured by counting all collisions that occur. It is the difference between the theoretically maximal number of collisions and the actual number of collisions c that occur when hashing n keys (equation 6).

$$f(x) = \frac{n^2 - n}{2} - c \tag{6}$$

The four fittest individuals out of the offspring and the fittest parent form the new parent generation. To prevent the fitness form decreasing from one generation to another, the fittest parent is always included in the survivor selection. The new four parents are the starting point for the new run through the evolution module. The used evolutionary operators in the evolutionary algorithm are reproduction and mutation. The crossover operator was set aside. As mentioned above, the evolution module consists of six functional blocks (Figure 9). An evolution cycle starts at the genome update module. This module holds the four individuals of the parent generation in a block RAM. It has an interface to the data path, to update the genome of the hash functions in the data path. A LFSR in the child select module selects twelve times one of the parents for the new offspring. This is done by random and without taking into account the differences in fitness of the parents. The selected genome is read out of the genome update module and transferred in double word portions to the mutate module. The mutate module consists in principle of 32 LFSRs. Every LFSR is responsible for mutating one bit of the genome part at the input of the module. The probability of the mutation of one bit is according to equation 6 between $\frac{1}{l}$ and $\frac{2}{l}$. After being mutated, the genome is used for configuring the hash function used for fitness evaluation. The fitness evaluation module computes the fitness of the actual genome. This is done by holding a memory that has a bit position for every entry of the hash memory. All existent keys are read out of the one memory in the data path that is not in use. Incoming keys are hashed and the

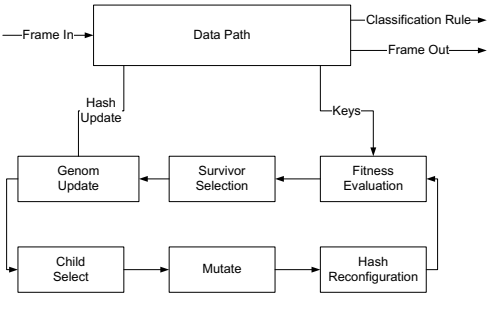

Fig. 9. Structure of the Evolution Module

memory is checked at the position of the hash value. If the memory indicates a free position, it is marked as used. Otherwise a collision counter is incremented and a new memory position is computed by the linear collision resolution. By counting all the collisions that occur when inserting all key to the memory the fitness is measured. After being evaluated the genomes of the offspring are transferred to the survivor selection module, where the four fittest ones are selected and transferred to the genome update module as the new parent generation.

5.3 Performance

The evolutionary algorithm especially the computation of the fitness of a hash function is extremely demanding regarding the computation time. During simulation four individuals, which produced an offspring of twelve, were evolved. To evolve 1000 generations with 100.000 keys with the SystemC model, a computation time of more than a day was needed on a 3.2 GHz machine. A software implementation of the evolutionary algorithm would probably have a computation time consumption comparable to the model. That is why the whole system was implemented in hardware, consisting of the data path and the evolution module. The most time consuming and thus performance critical module is the functional element evaluating the fitness. The fitness evaluation has a complexity of $O(N^2)$. If the initial hash function hashes all keys to the same value, $\frac{N^2-N}{2}$ collisions can occur when storing N keys to the memory. This is the worst case. But as the initial genome of the hash function is always chosen randomly, its quality is always better. The simulations showed that the initial hash functions produce at worst 20 million collisions for 2^{17} keys. The fitness evaluation module can compute one collision per clock cycle. Thus, for the above example of 2^{17} keys and an offspring of twelve, approximately 240 million clock cycles are needed. On a 125 MHz FPGA, the first generation would evolve in less than two seconds. The evolution rate increases rapidly with the hash function getting fitter. This is without any optimizations in the fitness evaluation. In Figure 10, the results of a simulation run with the ModelSim simulator are drawn. Here an evolution of 2048 32-bit wide keys has been performed. The keys were produced by a random generator. Running with 125 MHz, it took the system 1.84 seconds to evolve 1000 generations. As the fitness of the evolving system reached a very high value very fast, the graph is limited to twenty generations. The number of collisions occurring with the fittest individual was 187311 in the first generation (92.46 memory accesses per key) and after twenty generations limited to 923 (1,45 accesses per key). After the whole 1000 generations the number of collisions reached 845 (1,41 accesses per key).

5.4 Increasing the Computation Speed

There are different ways to increase the speed of the evolutionary process. As the time for fitness evaluation dominates the system computation time, the main attention regarding optimizations must refer to that hardware module. There are different ways for speeding up the fitness evaluation.

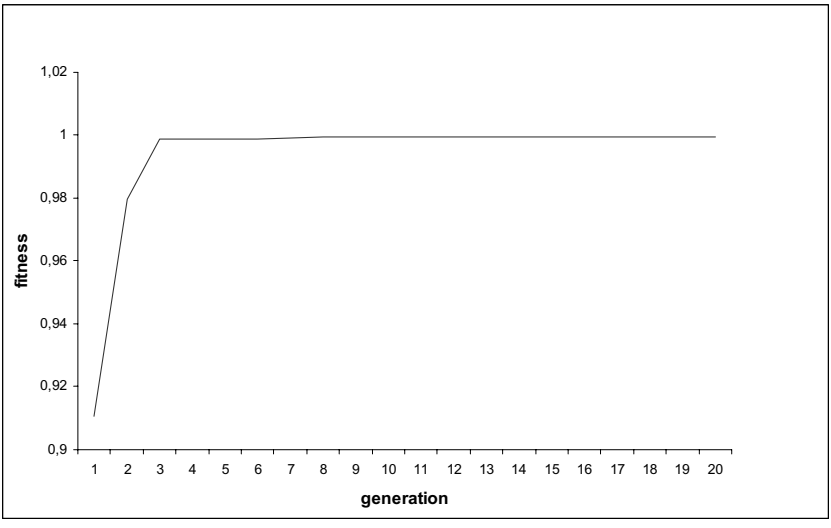

Fig. 10. Fitness of the ModelSim simulation of the packet classifier. The fitness is scaled to 1. Meaning a perfect hashing results in a fitness value of one.

One is the parallel implementation of one fitness evaluation element per offspring. This would increase the evaluation performance by a factor of twelve, as the fitness of all offspring would be computed at the same time. The computation time is bound to the offspring with the worst fitness. When looking for collisions while evaluating the fitness, a memory interleaving can be used. A linear collision resolve strategy is used. Thus, when the hash value of a key is known, all possible memory positions are known as well. The n-th possible position is at memory address $h(key) + n \cdot prime$. With that knowledge, many potential memory positions can be checked in parallel which speeds up the fitness evaluation process. A third way of speeding up the fitness evaluation is to stop evaluating when the number of collisions an offspring produces exceeds the number of collisions of the four best elements of the offspring. As only the four best elements of the offspring are selected for survival, the computation of an offspring element can be stopped, when it is clear, that the element is not among this group. This way the mean computation time for the fitness evaluation can be reduced.

In order to optimize the evolutionary algorithm, it might be useful to adapt the mutation rate according to the variance of the finesses of all the offspring. This method has been proposed in [7]. When the variance is high, the mutation rate should be decreased. When it is low, the mutation rate can be increased.

5.5 Implementation

The packet classifier was implemented into a Xilinx Virtex2 FPGA (XC2V4000-6-BF957). 2800 slices and 34 block RAMs are needed for the implementation of a packet classifier which can store 2048 classification rules with 32-bit keys and 16-bit rules. The implemented hash function is hash1.

6 Conclusion and Outlook

Both the simulations of the model and the implemented hardware showed that a packet classifier consisting of an evolvable hash function can be very efficient. The time complexity is roughly $O(1)$ and the memory demand is $O(N)$, even for very large rule sets. The actual used hash function is always designed for the momentary rule set by the hardware evolution. Evolving constantly, the hash function improves over time and adapts to changes in the rule set. These are excellent characteristics.

However, a drawback of the actual implementation is the limited range of application. At the moment the size of the key on which a rule search bases is configurable but still after implementation constant. That means, that for standard router applications, where longest prefix searches basing on keys with different lengths are the main application, the packet classifier is not very well suited. The mapping could only be done by setting the prefixes to the standard length. That would lead of course to huge redundancy. To map, i.e., a 28-bit prefix to a 32-bit key, 16 different keys with the same rule would have to be stored in memory. This problem can be solved by the use of multiple hash functions as presented in [8]. Here hash functions of different widths are used to do the prefix search. However, prefixes not matching any of the widths would still produce redundant entries.

The speed of evolutionary process has still to be increased in order to make faster adaptation to changing key sets possible. The four different possibilities that have been presented in this paper must be implemented in hardware. It has to be determined how the improvement of the fitness evaluation module can speed up the evolutionary process. Furthermore, the behavior of the packet classifier with real databases must be researched. At this moment only fabricated data basing on random functions has been used to demonstrate the behavior. Thirdly, the influence of the use of a small cache in the data path to solve the problem of numerous memory accesses for some keys must be tested.

A final evolving system could be implemented in a dynamically reconfigurable environment as mentioned in [9]. In such a system the hash functions would not need to consist of register controlled multiplexers. Instead there are just wires from input to output and some combinatorial logic. The wires are simply rerouted to evolve to a new generation. This is achieved by the FPGAs partial reconfigurability. The whole area of the FPGA, where the hash function is implemented, is reconfigured.

References

1. Gupta, P., McKweon, N.: Algorithms for packet classification. In: IEEE Network. (2001) 24–32
2. Gupta, P., McKweon, N.: Packet classification using hierarchical intelligent cuttings. In: IEEE Micro. (2000) 34–41
3. Goldberg, D.E.: Genetic algorthms in search, optimization, and machine learning. In: Addison-Weseley, 20th printing. (1999)

4. Yao, X., Higuchi, T.: Promises and challenges of evolvable hardware. In: IEEE Transactions on Systems, Man, and Cybernetics - Part C. (1999)
5. Knuth, D.E.: The art of computer programming, vol. 3 sorting and searching. In: Addison-Weseley, 3rd edition. (1998)
6. Tufte, G., Haddow, P.C.: Prototyping a ga pipeline for complete hardware evolution. In: Evolvable Hardware. (1999) 143–150
7. Damiani, E., Tettamanzi, A.G.B.: On-line evolution of fpga-based circuits: A case study on hash functions. In: Evolvable Hardware. (1999) 33–36
8. Broder, A., Mitzenmacher, M.: Using multiple hash functions to improve ip lookups. In: IEEE Infocom. (2001) 1454–1463
9. Kubisch, S., Hecht, R., Timmermann, D.: Design flow on a chip - an evolvable hw/sw platform. In: 2nd IEEE ICAC. (2005) 393–394