On the Impact of Caching for high Performance Packet Classifiers

Harald Widiger, Andreas Tockhorn, and Dirk Timmermann
Institute of Applied Microelectronics and Computer Engineering
University of Rostock, 18051 Rostock, Germany
Telephone: +49 (0)381 498-7276
Email: {harald.widiger;andreas.tockhorn;dirk.timmermann}@uni-rostock.de

Abstract

Hash functions have a space complexity of O(n) and a possible time complexity of O(1). Thus, packet classifiers exploit hashing to achieve packet classification in wire speed. Especially evolvable hash functions can adapt to a changing classification data base. But hash functions do have an important flaw. Some of the hashed keys may result in a large number of collisions. If those keys occur frequently, overall performance of a hash based packet classifier suffers. Although there is only limited or no existing locality in the data to be computed by a packet classifier, utilizing a cache can solve this problem. Unlike classical caches known from microprocessors, which are not adequate for packet classification and lookup algorithms, the proposed cache bases on a different strategy. It caches only the keys producing the most collisions instead of the ones that occur most often. That way, a cache improves the worst case performance of a hash function-based classifier. Based on simulation results, we show that even the mean performance improves significantly.

1 Introduction

Networking components, of which routers are the most widely known representatives, are the base of any digital communication network. Routers for example control all traffic flows by directing packets through the network. Their performance and features determine the services and quality of data communication. A router's main task is to route incoming data packets from input ports to proper output ports. To determine the correct output port, each packet needs to be classified based on its destination IP address. Other systems, e.g., intrusion detection systems, require a proper assignment of computation rules for each incoming packet as well. These and other functionalities are based on packet classification. Basically, a module processes every incoming packet depending on the results of a packet classification algorithm. In addition, high numbers of input/output ports, increasing bandwidths, and diffentiated Quality-of-Service (QoS) demands request classifiers to process incoming packets as fast as possible. Thus, designers have to construct classifiers, which satisfy two demands in order to assure high throughput and QoS:

- Find classification rules in huge data bases.
- Find them with very low latencies.

In general, to classify packets, a search algorithm of any kind is required. Usually, the search domain is much larger than the number of elements to be stored. Furthermore, the domain size may exceed the available memory capacity. Assume, for example, a packet classifier (router) with 2^{17}=131072 rules and packets with 32-bit wide keys, which could represent destination IP addresses. 2^{17} keys is realistic. The BGP routing database from [1] has approximately 260.000 entries. In this case, the search algorithm has to map 4,294,967,296 different values onto a new domain with 2^{17} entries. A direct mapping of IP addresses to the memory requires at least 4 GByte for the entries and would lead to a utilization of just 0.004%.

Therefore, hash functions can serve as search algorithm. A hash function $h(x)$ maps a value x onto a hash value, which is usually from a much smaller domain than the argument domain $(\mathrm{card}(\{h(x)\}) \ll \mathrm{card}(\{x\}))$. 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. That is referred to as collision. In a practical application, such collisions must be resolved. That can be done by simply adding a constant (preferably a prime number) $h(x) + p$ until a free memory entry is found. Thus, finding an entry requires $1 + collisions$ memory accesses.

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

Two ways of dealing with imperfection are:

- Optimize the hash function further, i.e., find a particular hash function $h(x)$ that maps all given n input values $x_{1..n}$ with as few as possible collisions. That is targeted by applying an evolvable hash function.
- Improve the performance of the memory lookup by using caches.

The remainder of the paper is organized as follows: In Section 2 the architecture and properties of the evolvable packet classifier (EPC) is explained. The cache's architecture is presented in Section 3. The simulation results of its implementation are presented in Section 4, before the paper concludes in Section 5.

2 EPC Architecture

Figure 1 sketches a simplified diagram of the EPC, that has been developed in previous research [2]. The hardware works as follows: A key parser extracts the key, e.g., the destination IP address, from an incoming packet and forwards it to the hash function by means of a switch. The hash function maps the key to a memory address, which is used for memory lookup. The result (classification rule) is forwarded to the actual routing unit. The routing unit is not part of the classifier itself. Because the memory interface has to resolve conflicts as well, it always compares the input key with that stored along each rule. In case of a collision, the interface resolves it by searching the memory linearly.

The hash function is realized in hardware (FPGA) and improves itself by applying a genetic algorithm (see Figure 2), which is sensitive to the current key set. Thus, it is an evolvable hardware. The hardware evolution model can apply variations of register values (genomes) to the hash function and can also evaluate its performance. The number of collisions denotes the performance of the hash function. It is thus used as quality criterion, i.e., the fitness of the evolutionary optimization.

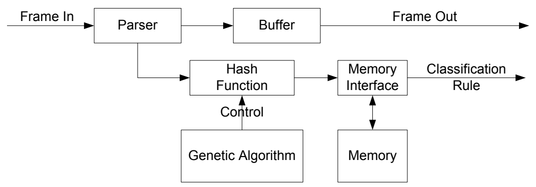

Figure 1. Simplified architecture of the EPC

2.1 Architecture of an Evolvable Hash Function

Each hash function is defined by a bit string S consisting of $s = 3\lg(n) \cdot m$ bits, with n denoting the number of bits to code the input values x and m denoting the number of bits to code the hash values $h(x)$. To hash, for example, 128k destination IPs, the values are $n = 32$ and $m = 17$. There a m blocks to code the hash values. Each block consists of three $n \rightarrow 1$ multiplexers to select bits from the input, which are inputs of an xor gate. Thus, the hash function uses three times $\lg n$ bits to determine the value of each of the m bits that code the hash values.

For the genetic algorithm, the task is to find an optimum in a search space with $s = 3 \cdot \lg(n) \cdot m$ dimensions, which is $s = 3 \cdot 5 \cdot 17 = 255$ in the example discussed above.

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. The interested reader is referred to the literature to get a better insight in the EPC's architecture [3], its hardware evolution model, and alternative evolvable hash functions [2].

2.2 Properties of the Evolvable Hash Function

The mean results obtained by the hash function are promising. As can be seen in Figure 3, the number of collisions that 32k keys cause after 2000 generations is at about 10650, on the average. In consequence, the system requires only 1.33 memory accesses per key to find an entry in the data base. However, as it is often the case with hash functions, some of the keys require many more than 1 or 2 memory accesses. Figure 4 reveals that the greatest majority of keys causes no (79%) or just one (13%) collision. But there are single keys that cause many more collisions. Simulations performed on the hardware prototype showed that in the worst case for some keys up to 23 memory accesses were required. A disproportionate appearance of those keys in

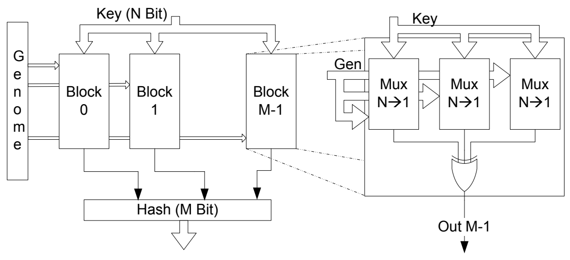

Figure 2. Architecture of the evolvable hardware hash function

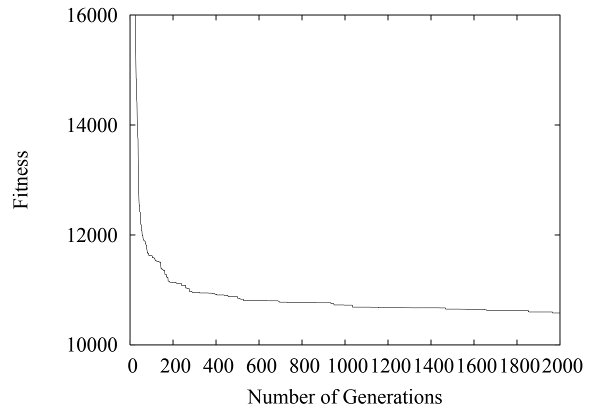

Figure 3. Performance of the evolvable hash function with 32k keys derived from real world data, a BGP routing data base [1]. The average value derives from 20 independent runs.

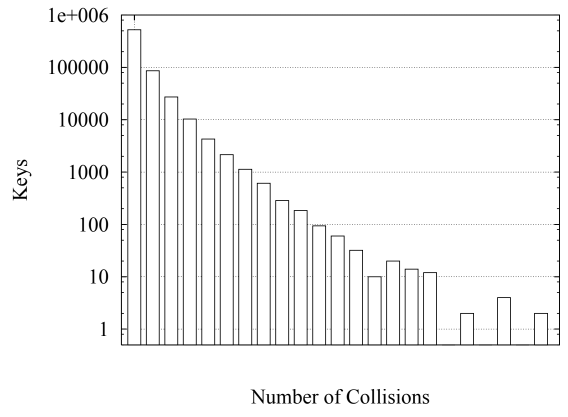

Figure 4. Distribution of collisions over 20 different sets of 32k keys without a cache

the data path affects the overall classification performance. Caching can solve that problem.

3 Caching in Communication Systems

Caching, especially in microprocessors, aims at reducing the gap between a fast processing unit and slow main memory. The cache works as fast buffer, which already holds data, the processing unit requires next. To do so, two characteristics of data are used:

- Spatial locality, meaning that data required in the future is located near data used at present.
- Temporal locality, meaning that data and/or code used in future is the same that is already used in present.

In communication, only temporal locality matters as data streams are computed.

3.1 Related Work

In [4] temporal locality of data in communication links has been discovered. Newman et. al. [5] on the other hand doubt that for faster links and modern networks. Especially in core networks with multi-Gbps links and thousands of connected users, sufficient temporal locality is probably not given as many different flows overlie each other.

Still, several authors use caching techniques to improve packet classification performance. In [6] the cache architecture of an Alpha processor is used to implement an efficient IP routing table lookup algorithm. Feldmeier [7] uses a cache successfully to improve the performance of a gateway router at MIT. Chen showed that caching effects lookup performance in an multimedia environment positively [8].

3.2 Cache Architecture

As mentioned above, there is probably no locality in multi Gbps links. Furthermore, the memory that stored the lookup data base is implemented as fast static RAM and can be accessed with the system speed. Caching a key, which does not produce a collision, would not increase the system performance. That is because a cache hit requires as much time as one memory access. Thus, the suggested cache does not try to hit as many keys as possible. It only caches the worst keys, meaning the keys, for which the hash function creates the most collisions.

The architecture of the cache is shown in Figure 5. The figure shows a simplified cache, which can compare two entries with the search key in parallel. This is referred to as 2-way associativity. The cache is implemented in parallel to the memory. Both cache and main memory start searching when a new key arrives. A cache hit only improves performance, if the searched key induces one or more collisions. In case there is no collision, a cache hit does not improve system performance.

In principle, the cache can be regarded as a table. The number of columns is determined by the desired associativity. Each column is implemented as a BlockRAM (BRAM) and a corresponding comparator. This architecture enables the cache to search a whole cache line in parallel. Hence, the number of cache lines is given by the cache size divided by associativity.

If the cache contains the searched key, exactly one of the comparators indicates the cache hit. The corresponding rule is transmitted to the memory interface. That shortens the

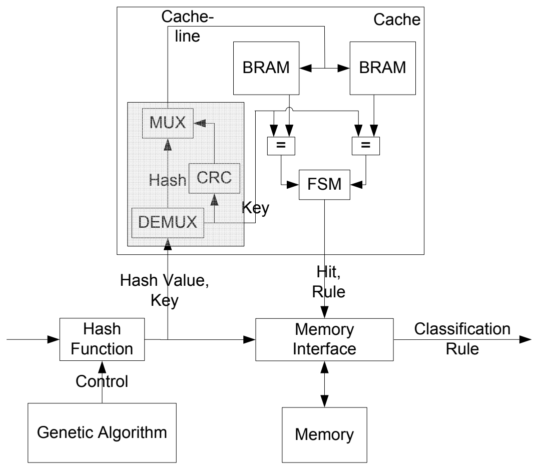

Figure 5. Simplified cache architecture

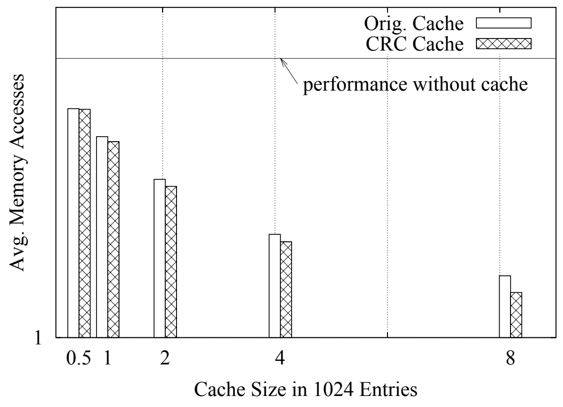

Figure 6. Performance of EPC with and without a cache implementation of different sizes and a constant 4-way associativity

search operation, because the memory interface does not have to resolve the collisions the key causes. Collisions have to be resolved only in case of a miss. In that case, rule, key, and the number of resolved collisions are transmitted to the cache, which tries to store the information in the corresponding cache line. This part of the cache is not illustrated in the figure. If the selected cache line is full, the key with the *fewest* number of collisions is exchanged by the new key, provided that the new one causes more collisions.

Size and associativity are the most important parameters for synthesis of the cache module. A third choice to be made is the mode for selecting the right cache line of a key. As illustrated in Figure 5, the line can be selected by a part of the regular hash value of the searched key or by an alternative CRC32 hash value. The way of determining the respective cache line, in which a key shall be stored, is of importance for the performance of the cache. The keys should be evenly distributed over the whole cache to ensure that as many bad keys as possible can be stored. CRC32 always generates well distributed hash values. They are independent from the state of optimization of the original hash function.

4 Cache Performance

To derive conclusions about the cache's quality, different experiments were performed. The influence of three decisive parameters size, associativity, and addressing mode was investigated.

4.1 Cache Size

The influence of the cache size was evaluated with a cache of 4-way associativity. Cache sizes between 512 and 8192 entries were evaluated. That means that the cache can store up to 1/4 of the existent entries. The upper bound for the cache size was given by the quantity of available BRAM within the selected FPGA platform (Xilinx Virtex4-FX20). Hardware costs in terms of logic resources are not influenced by the cache size. Figure 6 reveals the results obtained by twenty independent runs for each cache size. In each run 2 million randomly generated packets were sent through the data path of the EPC. The measuring was started after a setup time of 3 minutes, in which the hash function was optimized by the genetic algorithm.

As can be seen in the figure, data path performance increases remarkably. Without caching, on average 1.33 memory accesses are required to classify a key. With caching that number is reduced up to 1.05 memory accesses. Note that a lower bound of 1 memory access cannot be under-run. The improvement for a smaller cache is well reflected in the gain of the fitness value. When using a cache with just 1024 entries, the average fitness improves from 10650 to 7250 if the cache uses CRC32 hashing for addressing.

4.2 Associativity

The influence of associativity was investigated with a constant cache size of 1024 entries. Associativities from 1- to 8-way were evaluated. It is limited to 8-way because parallelism implied by an increasing associativity (see Figure 5) utilizes too many hardware resources of the FPGA. Figure 7 shows the performance gain for a cache depending on its associativity and the addressing scheme. As can be seen, the gain of high associativities is not significant. Whereas a higher associativity implies a great increase of resource utilization, meaning BRAM and logic resources. The most important finding is that it suffices to just implement the cache. Furthermore, already a 2-way associativity is enough to achieve a good cache utilization. Cache lines should al-

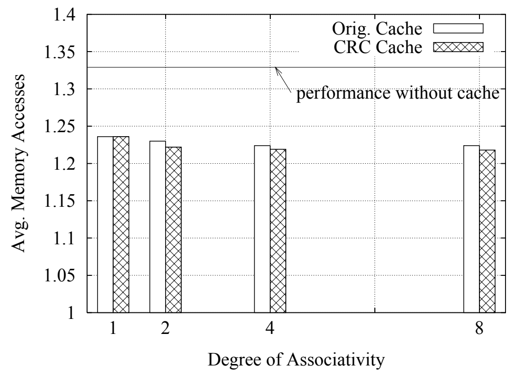

Figure 7. Performance of EPC with a cache of 1024 entries and different associativities

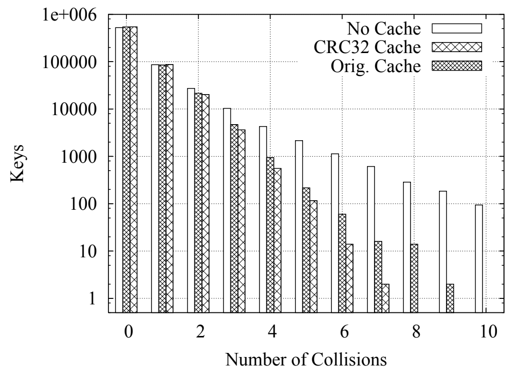

Figure 8. Distribution of the collisions with caches with a size of 1k entries and 2-times associativity (20 runs with 32k keys)

ways be addressed via CRC32 hash value of the key because this mode slightly increases the performance and consumes with just 32 slices hardly no additional logic resources.

4.3 Worst Case Performance

While the above evaluations target the mean performance, the more important point for a dependable hash based classifier is the worst case performance. As derives from Figure 4, the hash function produces up to 22 collisions. As Figure 8 reveals, with caching that value is reduced. With ordinary addressed cache, all simulation runs resulting in not more than 9 collisions. CRC32-addressing produces even better results. The maximum is at 7 collisions and the number of keys with more than three collisions is reduced even further.

5 Conclusion

We showed that a cache is capable of improving mean and worst case performance of a hash-based packet classifier. Therefore, we presented a cache, which caches only critical keys producing many collisions instead of those that occur often. That way, the cache's performance does not depend on temporal locality in the data. We analyzed the influence of both cache size and the degree of associativity on the cache performance in order to be able to find performing configurations. Furthermore, the influence of the use of CRC32 as alternate hash function to address the cache was determined. The results show that even a simple cache with just 2-times associativity and a size that stores just 3% of the overall keys improves mean performance of the classifier by as much as 8%. More important is that the cache limits the maximal number of required memory lookups for the worst keys from 23 to 8.

References

[1] "BGP routing data of an internet core router." [Online]. Available: http://archive.routeviews.org/oix-route-views

[2] H. Widiger, R. Salomon, and D. Timmermann, "Packet classification with evolvable hardware hash functions - an intrinsic approach," in *Second Interational Workshop on Biologically Inspired Approaches to Advanced Information Technology (BioADIT 2006)*, ser. LNCS 3853, January 2006, pp. 64–79.

[3] R. Salomon, H. Widiger, A. Tockhorn, and D. Timmermann, "Rapid Evolution of Time-Efficient Packet Classifiers," in *Proc. 2006 World Congress on Computational Intelligence to be held in Vancouver*, Jul. 2006.

[4] C. Partridge, "Locality and route caches," in *Proc. of NFS Workshop on Internet Statistics, Measurements, and Analysis*, February 1996.

[5] P. Newman, G. Minshall, T. Lyon, and L. Huston, "Ip switching and gigabit routers," 1997.

[6] T. Chiueh and P. Pradhan, "High performance IP routing table lookup using CPU caching," in *INFOCOM (3)*, 1999, pp. 1421–1428. [Online]. Available: citeseer.ist.psu.edu/309968.html

[7] D. C. Feldmeier, "Improving gateway performance with a routing-table cache," in *Proc. of the Seventh Annual Joint Conference of the IEEE Computer and Communcations Societies, IEEE INFOCOM*, 1988, pp. 298–307.

[8] X. Chen, "Effect of caching on routing-table lookup in multimedia environment," in *Proc. of the Tenth Annual Joint Conference of the IEEE Computer and Communications Societies, IEEE INOFCOM*, April 1991, pp. 1228–1236 vol.3.