BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

PROBABILISTIC PACKET CLASSIFICATION ACCELERATION ON FPGA
AKCELERACE STATISTICKÉ KLASIFIKACE PAKETŮ POMOCÍ FPGA

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR DENIS KURKA
AUTOR PRÁCE

SUPERVISOR Ing. LUKÁŠ KEKELY, Ph.D.
VEDOUCÍ PRÁCE

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

148209

Ústav:	Ústav počítačových systémů (UPSY)
Student:	**Kurka Denis**
Program:	Informační technologie
Specializace:	Informační technologie
Název:	**Akcelerace statistické klasifikace paketů pomocí FPGA**
Kategorie:	Vestavěné systémy
Akademický rok:	2022/23

Zadání:

1. Prostudujte existující metody pro klasifikaci (filtraci, vyhledávání) síťových paketů. Zaměřte se primárně na třídu statistických (ne-exaktních) metod.
2. Seznamte se blíže s fungováním alespoň tří metod této kategorie.
3. Vyberte jednu z metod nebo navrhněte jejich vhodnou kombinaci pro implementaci v FPGA.
4. Implementujte zvolenou/navrženou metodu v jazyce VHDL.
5. Zhodnoťte parametry vytvořeného FPGA modulu v různých nastaveních z pohledu spotřebovaných zdrojů, latence a propustnosti.
6. Zhodnoťte dosažené výsledky a diskutujte možnosti pokračování projektu.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:
Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Kekely Lukáš, Ing., Ph.D.**
Vedoucí ústavu:	Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání:	1.11.2022
Termín pro odevzdání:	10.5.2023
Datum schválení:	31.10.2022

Abstract

Classifying network packets is a crucial task in networking systems, as it allows for efficient routing and filtering of data. Probabilistic filters are a classification method that uses different techniques to approximate the membership of a packet in a set of rules. This work investigates three algorithms: Bloom, cuckoo, and xor filter. The main aim is to compare the performance of these three methods when implemented as hardware components in FPGA systems. The evaluation criteria include error rate, maximal frequency, and FPGA resource usage, primarily focusing on memory. The results indicate that the xor filter outperforms the others regarding error rate, which is superior in any error rate category. The Bloom filter is the fastest option for smaller and quicker components where a higher error rate is tolerable. The cuckoo filter is the most resource-efficient when FPGA logic is the primary concern. These findings contribute to the development of optimised classification systems and provide valuable insights into the possibilities of implementing probabilistic filters in hardware architectures.

Abstrakt

Klasifikace síťových paketů je klíčovým úkolem v síťových systémech, protože umožňuje efektivní směrování a filtrování dat. Pravděpodobnostní filtry jsou klasifikační metoda, která používá různé techniky k aproximaci členství paketu v sadě pravidel. Tato práce zkoumá tři algoritmy: Bloomův filtr, cuckoo filtr a xor filtr. Hlavním cílem je porovnat výkon těchto tří metod při implementaci jako hardwarové komponenty v FPGA systémech. Kritéria hodnocení zahrnují chybovost, maximální frekvenci a využití zdrojů FPGA se zaměřením na paměť. Výsledky ukazují, že xor filtr překonává ostatní v oblasti chybovosti, ve všech kategoriích. Bloomův filtr je nejrychlejší volbou pro menší a rychlejší komponenty, kde je vyšší chybovost tolerovatelná. Cuckoo filtr je nejefektivnější z hlediska využití FPGA logiky. Tyto poznatky přispívají k vývoji optimalizovaných klasifikačních systémů a poskytují cenné informace o možnostech implementace pravděpodobnostních filtrů v hardwarových architekturách.

Keywords

packet filtering, FPGA, hardware architecture, cuckoo filter, Bloom filter, xor filter

Klíčová slova

filtrování paketů, FPGA, hardwarová architektura, cuckoo filtr, Bloom filtr, xor filtr

Reference

KURKA, Denis. *Probabilistic Packet Classification Acceleration on FPGA*. Brno, 2023. Bachelor’s thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Lukáš Kekely, Ph.D.

Probabilistic Packet Classification Acceleration on FPGA

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Lukáš Kekely, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Denis Kurka
May 8, 2023

Acknowledgements

I want to thank my work supervisor, Ing. Lukáš Kekely, Ph.D., for his valuable advice and suggestions, which immensely helped me create this work.

Contents

Chapter 1

Introduction

Packet classification is crucial in networking, as it allows the packets to be routed, prioritised or blocked. Furthermore, with the growing amount of data, it becomes increasingly essential to study algorithms that deal with this problem efficiently.

The classification boils down to comparing an element against a set of rules and finding an appropriate match. This is commonly done exactly, using various approaches such as linked lists or tree data structures. However, with the ruleset increasing in size, it may be better to sacrifice perfect matching for better space efficiency. Ideally, this introduced error should only be a false positive; if the result of the match is false, it is undoubtedly not in the ruleset.

One of the most well-known approximate set membership algorithms is a Bloom filter [1]. This algorithm is known for its speed and ability to add a new rule to an already constructed filter. It has many variations [16] and can even be implemented in hardware [17]. Other methods include a cuckoo filter, capable of adding and removing rules [6]. On the other hand, xor probing algorithms, which cannot be changed after construction, can achieve the best memory efficiency [7, 8, 4].

As only the hardware implementation of Bloom filters was studied, this work focuses on implementing Bloom, cuckoo and a xor filter in FPGA design. The resulting components should reveal the merits and drawbacks of the presented algorithms, not limited to their construction speed and memory efficiency. But also their resource usage and throughput in hardware design.

The thesis begins with a short outline of computer networks and packet classification, followed by a theoretical description of exact and probabilistic techniques, focusing on probabilistic. Three algorithms are then introduced Bloom, cuckoo and xor filter, all described in detail and followed by their design and implementation on an FPGA. Finally, the implemented designs are compared against each other.

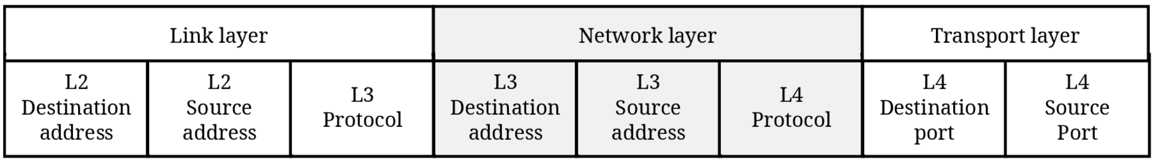
Chapter 2

Packet classification

2.1 Networking and packets

When a device wants to send data over the network, it encapsulates this data into small segments called packets and transmits them. The computer network is made of routers that then route the sent packets to their destination. To correctly get everything to its destination, routers make forward decisions by comparing the destination address to their forwarding table. Also, packets can get different treatments, some can get prioritised, and others can be dropped entirely. Hence, routers need to have a way to distinguish between packets [5]. This ability to differentiate packets is called packet classification.

The classification is based on attributes in the packet header. This header is created by gradually encapsulating higher layers to lower layers of the network model, adding different information about the packet. Application data are first encapsulated with the transport layer, creating either a TCP or UDP packet with its destination and source ports. The IP header is then attached to this data layer with the appropriate source and destination IP address. Finally, this IP datagram is inserted into the appropriate network interface layer header and receives its address for this layer [11]. Each header stores many more values about the packet, but they are only rarely used when categorising packets. A simplified packet header with some of the fields used for classification is shown in figure 2.1 [5].



Link layer			Network layer			Transport layer	
L2 Destination address	L2 Source address	L3 Protocol	L3 Destination address	L3 Source address	L4 Protocol	L4 Destination port	L4 Source Port

Figure 2.1: Example packet header. Only layers with interesting values for packet classification are shown.

As was said, it is frequently the case that we want to select only a specific subset of packets based on the packet header attributes. To achieve this packet header is compared to a ruleset, and based on membership in this ruleset, packets are categorised. However, packet classification can be performed simultaneously on one or multiple header fields, creating a multi-dimensional problem [14]. Moreover, with the increasing demand for performance, simple algorithms such as sequential search are hardly used. Therefore, the need for faster and more space-efficient algorithms forms.

2.2 Classification methods

With the increasing number of rules in the classifier and the need to search with multiple fields simultaneously, the classification algorithms face a significant challenge [9]. Classification methods use a given set of rules to construct a filter which is then used to find matches in incoming packets. Different approaches are used to achieve this.

Today's techniques construct an optimised data structure from the ruleset and then try to match the searched attributes to the rules [10]. As these methods search with all the rules, they will always respond correctly. We can label them as exact methods.

In contrast, algorithms that do not store all the rules wholly also exist, saving on performance and space. However, they also introduce a small error rate when querying this data structure. These are called probabilistic methods.

Exact methods

The main idea is to search the rules with the packet header attributes until we get an exact match for all the values. The rules are saved in different data structures allowing for fast and efficient search [10].

One of the most straightforward data structures is sorted linear search. A sorted linked list is created from the ruleset, and every incoming packet is compared to each element sequentially. Searching through a linked list has a linear time complexity; therefore, a linear search must also have a linear time complexity [14]. However, given that the ruleset can contain hundreds of thousands of rules, a more efficient solution is needed.

Algorithms like the trie structure try to combat this by traversing through a tree data structure, lowering the time complexity. Hardware-oriented algorithms like Ternary CAM exploit parallelism on hardware to implement multi-dimensional classification allowing for constant time complexity [5]. Other algorithms based on geometry perspective and heuristics also exist.

Probabilistic methods

It is possible to allow a representation of the ruleset that only stores each rule partially. In other words, the filter becomes lossy. With this, lossy filters dramatically improve storage requirements. This feature of incomplete filters is the main idea behind probabilistic methods [16].

The probabilistic nature of the filter can be implemented in many different ways. However, all implementations aim for the same, a filter with a size approaching the informational theoretic lower bound [12], ideally with the ability to dynamically modify elements and with the least error rate possible.

Minimising the filter size is tricky, as with the data size closing on the theoretical bound, the algorithms tend to get more complex and have less dynamicity. The most space-efficient algorithms generate only static filters, where nothing can be added or removed after creation [4]. The dynamic filters often only implement dynamic adding to the rules, with exceptions even allowing for dynamic removal.

The probabilistic filter must also ensure that the negative result, not in the ruleset, is always correct and that only the presence of the queried element in the ruleset is occasionally incorrectly assumed. In other words, false positives are acceptable, but false negatives are not.

Many methods satisfy this requirement. Burton Bloom, in 1970 [1] described a method called Bloom filter that is the foundation of many methods currently used. Cuckoo filters make use of cuckoo hashing to construct a probabilistic filter. Moreover, recently proposed methods use xor probing to construct static filters with high space efficiency. These methods and more will be described in detail in the next chapter.

Chapter 3

Probabilistic methods

3.1 Bloom filter

Bloom filter is a simple method of querying an element in a set with an allowable error, described by Burton H. Bloom in 1970 [1]. A Bloom filter provides a compact representation of a set and two operations, element query and element insert. When querying an element in the Bloom filter, the query either returns “definitely not in set” or “possibly in the set”. The more elements are added to the Bloom filter, the higher the false positive rate will become. However, the Bloom filter can be adjusted to guarantee a specific false positive rate until a given number of items are inserted. This results in an all-around filter that can be used in various situations.

Detailed description

Bloom filter is an array of m bits initially set to zero. The main idea is to encode an element by uniformly setting k bits to one. The positions of k bits in the array are calculated using k hash functions.

When querying for an element, if one of the k bits is zero, the element is undoubtedly not in the set. If all of the queried bits are set to one, an element could be part of the set but does not have to be. One can imagine a short array with too many inserted elements, resulting in all bits being set to one. By querying for any element, the query will always return a positive result, even if the element is not in the set. In contrast, if an array is blank and all bits are set to zero, any query will result in a negative result.

Figure 3.1 presents a practical example of inserting an element into a Bloom filter. Firstly all k hash functions are calculated for the inserted element. Then indexes to the bit array are obtained by taking array size modulo from each hash and writing ones to each index in the array.

Element query is similar to inserting an element. First, elements are hashed with k hash functions, and then positions are calculated in the array. Lastly, if at least one bit in calculated positions is zero, it is not in the set. If all bits are one, the queried element will likely be in the set.

As stated many times, queried elements are only likely to be in the set, never absolutely in the set. This is the weak point of Bloom filters, called false positives, in other words, elements that are reported to be in the set but are not in the set. An example of a false positive is shown in figure 3.2. Elements A and B are inserted into the Bloom filter, but a query for element C results in success, even if C is not part of the set.

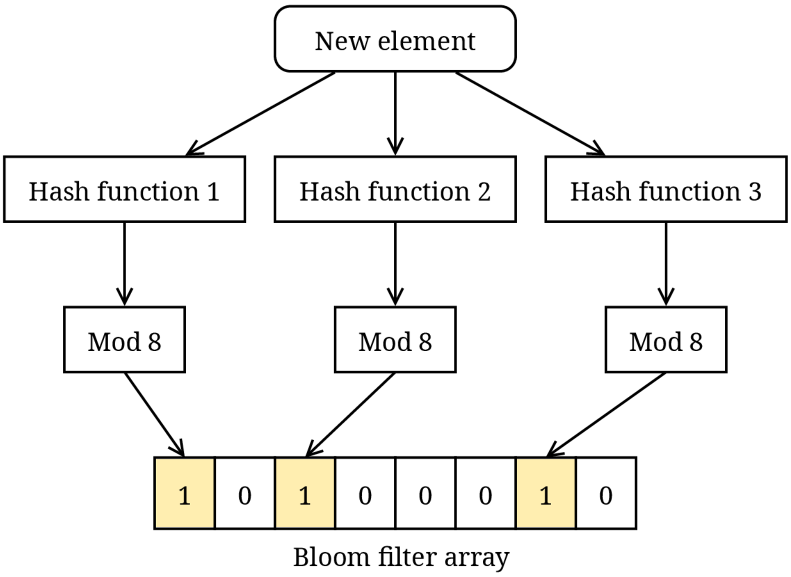


Figure 3.1: Bloom filter insert example.

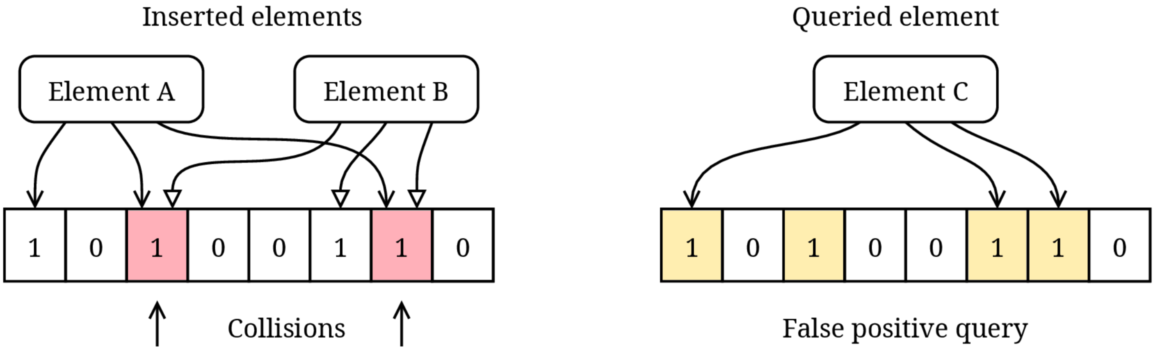


Figure 3.2: False positive query.

As shown in figure 3.2, a collision happens when the same index is calculated for different elements, writing one to the same place in the array. The collisions are the main reason a Bloom filter cannot remove elements. If we would try to delete an element, similarly to inserting a new element, we could set a common bit of multiple elements to zero and unintentionally remove multiple elements.

Allowable errors

One of the most beneficial features of Bloom filters is that we can calculate the false positive rate ϵ based on the number of inserted elements n, the array size in bits m and the number of hash functions k, given by the following formula number (3.1). All following formulas (3.1)(3.2)(3.3) are taken from the analysis of the Bloom filter in [16].

$$\epsilon = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1 - e^{-kn/m}\right)^{k} \tag{3.1}$$

We can minimise the probability of false positives for the number of hash functions k in formula (3.1), leading to a formula number (3.2), fixing the number of hash functions based on the array size m and the number of elements n.

$$k_{optimal} = \frac{m}{n} \ln 2 \approx \frac{9m}{13n} \tag{3.2}$$

With the number of hash functions fixed by the array size and the number of inserted elements, it is possible to calculate the ideal array size of the Bloom filter given only the number of inserted elements n and the desired false probability p, as shown in formula number (3.3).

$$m = -\frac{n \ln p}{(\ln 2)^2} \tag{3.3}$$

It is necessary to note that this analysis is optimistic and only suitable for large Bloom filters [16].

Bloom filter variants

The standard Bloom filter is simple and powerful. However, many other variants exist, improving various characteristics of Bloom filters, including deletion support, improved space efficiency and more. For a good summary of Bloom filter variants, consult [16].

In a standard Bloom filter, numerous hash functions index one array. This can be a problem in some situations. For example, if operations on the Bloom filter are parallelised, it is essential not to read and write on the same memory simultaneously. We can split the array into k blocks, each block for each hash function. With this simple change, race conditions are mitigated.

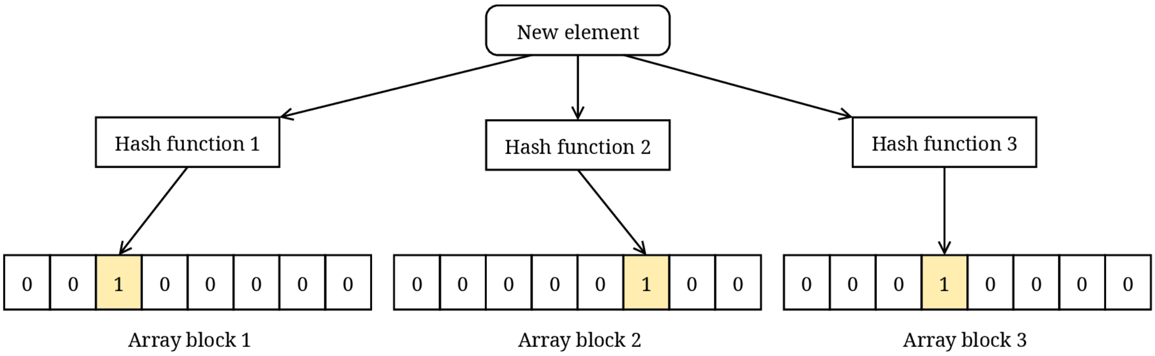


Figure 3.3: Split Bloom filter insert example.

Overall, Bloom filters are fast, small, and easily parallelised. Capable of querying and inserting elements without removing capability. The following methods often address Bloom filters and try to surpass them in various ways.

3.2 Cuckoo filter

A cuckoo filter is a probabilistic set membership method based on a cuckoo hash table [13] storing only the fingerprint of inserted elements instead of the elements themselves. The size of the fingerprint is determined by the target false positive probability. The cuckoo filter table can be filled with fingerprints up to 95%, reaching high space efficiency. A cuckoo filter also supports the removal of inserted elements. Authors of this method claim that a cuckoo filter is practically better than a Bloom filter based on its space efficiency and support for removals [6]. First, cuckoo hashing is described as the foundation of the cuckoo filters, and then the cuckoo filter itself is presented.

Cuckoo hashing

Cuckoo hashing is a technique to eliminate collisions in a hash table. A cuckoo hash table consists of two independent tables. When inserting an element into a cuckoo hash table, the inserted element is hashed using two hash functions, one for each table. The element is then inserted into one of two hash tables. If the element cannot be inserted into any table, the existing element in the first table is removed and inserted into the other table using the appropriate hash function. If the removed element has a collision in the other table, we can repeat this process until both tables are reorganised [13]. A maximum of relocations for one inserted element is set beforehand to mitigate an infinite loop. If the maximum of relocations is passed, the table is considered full.

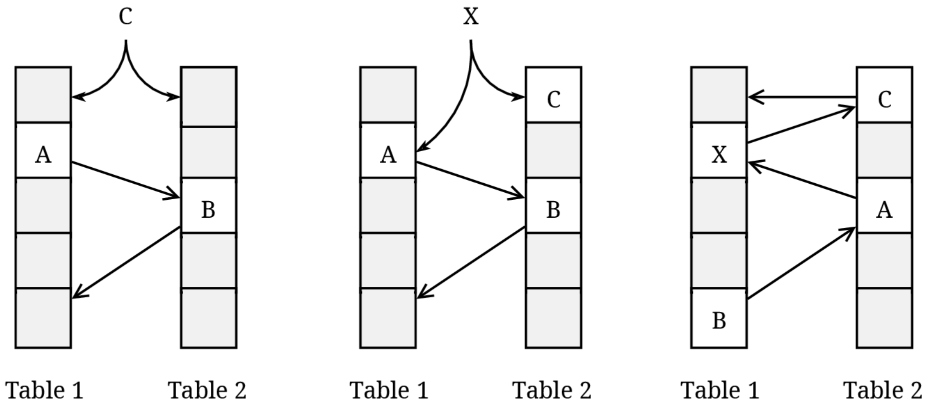


Figure 3.4: Cuckoo hash table insertion. Arrows show alternative element locations.

To query an element in the cuckoo hash table. Firstly hash the element with both hash functions and check for the presence in both tables. If the element is not present in any of the tables, it is naturally not in the set. If the element is in one of the two tables, it is in the set.

The number of tables in the cuckoo hash can be more than two. We can implement multiple tables, each with its hash functions. Alternatively, we can simplify the cuckoo hash table by using only one table with two or more indexing hash functions. The cuckoo filter uses this approach.

Detailed description of cuckoo filter

Due to being based on a cuckoo hash table, a cuckoo filter is similar in many aspects. The cuckoo filter also uses two hash functions to determine an element's position in the table. Moreover, it also has to reorganise the table similarly if a collision is encountered. However, instead of storing elements themself in the table, the cuckoo filter stores only a fingerprint, which is also a result of a hash function. The fingerprint is usually very short, never surpassing more than 8 bits. In cuckoo filters, an inserted fingerprint is called an entry, and in contrast to cuckoo hash tables, one address in a cuckoo filter table can store multiple entries, called a bucket. A bucket usually contains 4 or 8 entries [6]. To summarise, a cuckoo filter table is an array of buckets that individually store entries called fingerprints.

Inserting an element into a cuckoo filter consists of three steps. Firstly, indexes to both buckets $h_1(x)$ and $h_2(x)$ are calculated using hashing scheme described in [6] as follows:

$$h_1(x) = \text{hash}(x), \tag{3.4}$$
$$h_2(x) = h_1(x) \oplus \text{hash}(x\text{'s fingerprint}). \tag{3.5}$$

Secondly, if both buckets have an empty entry, one of the two buckets is chosen, and the fingerprint of the new element is inserted in a random entry in the bucket. If only one bucket is empty, the fingerprint is stored in a random entry in this bucket.

Thirdly, if both buckets are full, one of the two buckets is chosen, and a random entry in that bucket will be relocated. It is possible to compute the alternative location j of the inserted element only using its fingerprint and the current location i as shown by:

$$j = i \oplus \text{hash}(\text{fingerprint}). \tag{3.6}$$

Reorganising the table without needing the original element is possible with the above property. Note that this substantial property is only possible due to the xor operation when calculating the location of the buckets in (3.4). One can notice that the fingerprint is hashed again, even though the fingerprint is a result of a hash function. If we use the fingerprint without the second hash, the two calculated locations will land close to each other in the table. This is an unwanted behaviour that lowers uniformity in the table, therefore increasing the probability of collisions. Thus a hash of the fingerprint is used. Figure 3.5 shows an example of inserting an element into the cuckoo filter.

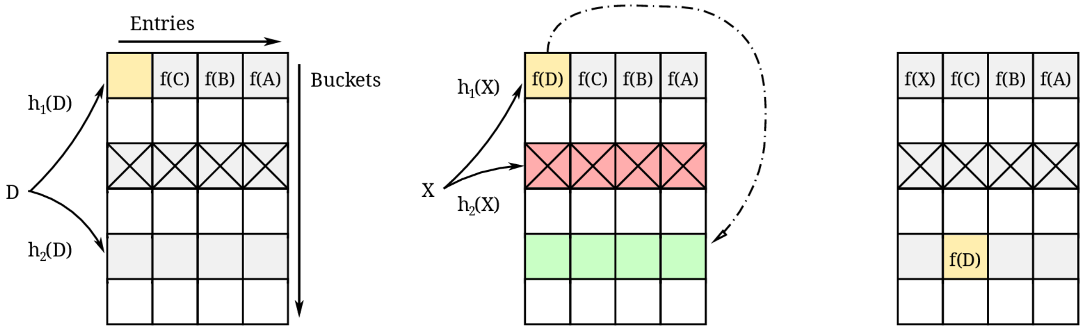


Figure 3.5: Cuckoo filter insert example. Crossed entries are full, and yellow-coloured entries are random decisions.

An element query is the same as in a cuckoo hash table. With two bucket indexes calculated, every entry in both buckets is then searched for the fingerprint of the queried element. If the fingerprint is not found, the queried element is not in the set. Given that the fingerprint is only a compact representation of the original element, if the fingerprint is present in one of two buckets, it is only likely that the queried element is in the set.

To delete an element from the cuckoo filter. First, we locate the desired element by performing a query operation. Then we remove a copy of a found fingerprint. It is important to note that two elements can hash to the same fingerprint and reside in the same bucket. With that in mind, only one fingerprint copy must be deleted. For example, elements X and Y hash to the same fingerprint and are stored in the same bucket. If we would perform a delete operation on element X and delete both fingerprints, a query for element Y would result in a false negative. Therefore only one fingerprint copy is deleted. The critical consequence is that after the deletion of one fingerprint, a query for element X will still return a positive result, which is, of course, a false positive [6].

In summary, a cuckoo filter is an approximate set membership method improving on Bloom filters by supporting dynamic deletions, better query performance and better space efficiency for applications with low false positive probability as stated by the authors [6].

3.3 Xor probing filters

All filters using hashing can be categorised into three groups:

- And probing filters: An element is found when all locations match. A Bloom filter is an example of this.
- Or probing filters: An element is found when one of the locations matches. A cuckoo filter is an example of this.
- Xor probing filters: An element is found when a bitwise xor of all locations is a match.

A Bloom filter and a cuckoo filter are both dynamic filters capable of changing elements in the filter. In contrast, xor probing is only known to work with static filters, incapable of adding or removing elements. However, xor probing filters can achieve better space efficiency than any other dynamic counterpart. The key idea is to match a fingerprint from the queried element to a bitwise xor of values in hashed positions of an array. Xor filter [7], binary fuse filter [8] and a Ribbon filter [4] are all implementations of a xor probing filter.

Xor filter

The xor filter is an array B containing the k-bit values calculated during the filter's construction from the inserted elements. Three hash functions h_0, h_1, and h_2 are chosen during the filter construction to index in the array B. A fingerprint function is used to query an element in a filter. The fingerprint function is assumed to be independent of the h_0, h_1, h_2. The longer the fingerprint is, the lower the false-positive probability of the xor filter. In most applications, keys are between 8 to 20 bits [7].

An element is likely in the xor filter if the following equation is satisfied:

$$B[h_0(x)] \text{ xor } B[h_1(x)] \text{ xor } B[h_2(x)] = \text{fingerprint}(x). \tag{3.7}$$

In other words, to query an element in the filter, values from the array B indexed by the hash functions h_0, h_1, and h_2 are aggregated using the xor function and compared to the fingerprint of the queried element. If the xor aggregation reconstructs a fingerprint, the element is likely in a set. Otherwise, it is not in the set. For this query operation to work properly, we must ensure that the equation (3.7) holds for every element in the xor filter.

This is ensured in the construction of the filter. The construction of the xor filter can be split into two parts:

- The mapping step: Appropriate hash functions and the order in which we must calculate the values in the array B are found.
- The assigning step: The values of the array B are calculated and inserted into the array.

Firstly three hash functions are picked randomly and independently from the fingerprint function. The mapping step is then performed with the hash functions and the set of elements that we want to contain in our filter.

The key idea behind the mapping step is to find out if the three chosen hash functions can ensure the property (3.7) on the set of inserted elements. If so, then for each element x, an index i is found. This index i will result from one of the three hash functions. For each index i, the assigning step then computes a value according to:

$$B[i] \leftarrow \text{fingerprint}(x) \text{ xor } B[h_0(x)] \text{ xor } B[h_1(x)] \text{ xor } B[h_2(x)]. \tag{3.8}$$

However, from the equation (3.8), we can see that the value $B[i]$ must be calculated before it will be used in a different calculation as a $B[h_0(x)]$, $B[h_1(x)]$ or a $B[h_2(x)]$. The mapping step must also ensure this property and therefore returns the pairs of elements and indexes ordered so that all values are known.

The mapping step can fail, and three new hash functions must be picked. This cycle repeats until we find three hash functions that satisfy our needs. The probability of success rate is estimated to be greater than 80%. To achieve this probability of success, the array B must be larger than the number of the inserted elements, approximately 1.23x larger.

After the mapping and the assigning step are done, the xor filter is successfully constructed. Naturally, insert and remove operations are impossible, as writing in the array B would likely change the resulting aggregated xor used to compare with fingerprints.

Only a general idea of the construction algorithm is described; details regarding the construction of the xor filter can be found in [7].

Other implementations of xor probing filters

The construction of the xor filter is very demanding and can take approximately 4x longer to construct than to fill a Bloom filter with the same elements [7]. Also, the space overhead for the xor filter could be improved by different methods.

Improved implementation of the xor filter called binary fuse filter exists, improving the construction time and the storage requirements. The construction algorithm is similar to a xor filter but simpler [8].

Another recent implementation of xor probing filters is a ribbon filter. It also improves the construction time and storage requirements. The construction algorithm solves a linear system with their proposed solver to construct the ribbon filter [4].

Chapter 4

Design

Newly proposed filters often describe the filter algorithm only theoretically, and the example implementation, along with the target implementation, is usually only in the software domain.

In contrast to the software solutions, the hardware implementation also needs to consider the complexity of the filter and the resulting usage of the FPGA circuitry, which means that faster and better filters can result in practically inferior filters. Moreover, newly proposed xor probing filters are only implemented in software applications.

A design was developed for Bloom, cuckoo, and xor filters with a shared environment to explore the usage of hardware filters further. The final design and implementation surpassed the scope of the original thesis assignment as three solutions were created. Moreover, all techniques were created based on the specified algorithms and were refined for optimal hardware implementation. The following section describes the design of these developments.

4.1 Bloom filter design

Due to the Bloom filter's simple algorithm and parallel nature, the Bloom filter has been successfully implemented many times in hardware applications [15][17] and is used not only in packet classification.

The previous research on the hardware implementations of the Bloom filter mainly studies the optimisation of the hash functions used in the filter. Many propose new hash function algorithms for efficient calculation and function generation. Hash function generation is essential to achieve the best result in filtering, as the number of hash functions k cannot be constant and should be changed based on the number of maximal inserted elements n and the cumulative array size m according to $k_{optimal} = \frac{m}{n} \ln 2 \approx \frac{9m}{13n}$. Due to this, a hash function generator is designed to adjust the number of hash functions on the filter's creation.

One way of generating hash functions is double hashing. Double hashing implements two independent hash functions, multiplying one of the outputs by any function and then summing the outputs. By altering the parameters of the multiplying function, we can generate any number of hash functions [16]. Another option is to implement k number of the same hash function, and then bit shift the input to each hash function so that each hash function has different input, as shown in figure 4.1.

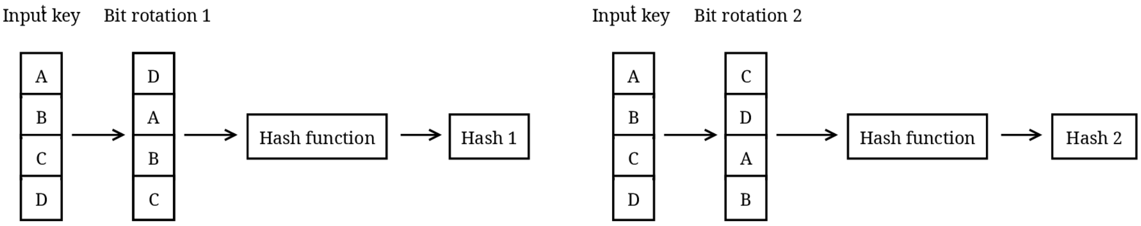


Figure 4.1: The hash functions generation example.

The latter option was chosen for the final design as it seems more straightforward and efficient for hardware implementation. However, it is essential to note that the Bloom filter can need ten or more hash functions, which can be expensive to implement.

Another critical part of the Bloom filter is memory access. The design is a variant of the classic Bloom filter that splits the memory array into multiple arrays, one for each hash function. Splitting the memory is essential as each hash function will be computed in parallel along with memory access. This approach is also used by the existing implementations of the Bloom filter [15][17].

The whole Bloom filter design then functions as follows. The inputs and outputs of the design are:

- Input key: String of bits to be searched or written.
- Match: The key is present in the filter.
- Configuration interface: Used to change memory values in the filter externally.

When the input is a valid key, indexes for every memory block are calculated by rotating, hashing and performing the modulo operation. The values at each index are read and accumulated with an and operation. The result is then written to the match signal output. This figure 4.2 reflects this behaviour. From the block diagram is also clear that this reflects the split Bloom filter variant shown in the Bloom filter variants description 3.1.

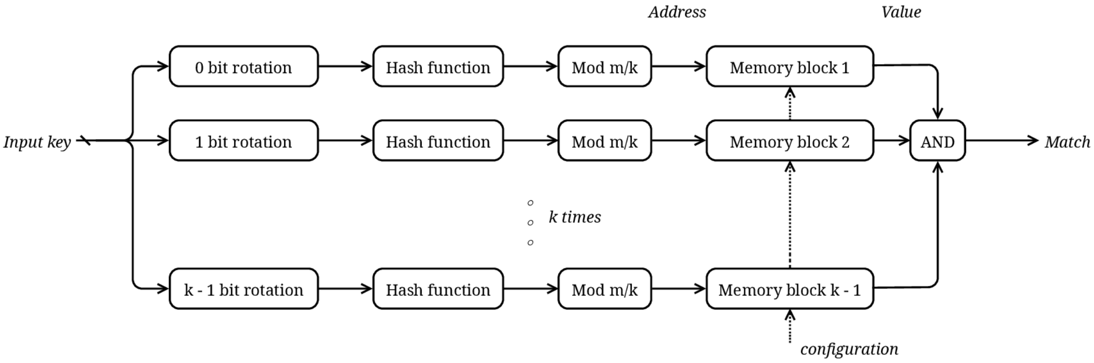


Figure 4.2: The Bloom filter block diagram.

To sum up, the design consists of k number of pipelines for calculating the index with a memory block, input and output logic. Input bit rotation creates a new hash function for each pipeline when needed.

4.2 Cuckoo filter design

As previously mentioned, the cuckoo filter streamlines cuckoo hashing by employing just one table rather than multiple hash tables with two hash functions. Nonetheless, accessing the same memory can be problematic in hardware applications. Consequently, the memory was split into two identical tables, one for each hash function.

Since this modification divides the filter's memory model, it is not expected to impact the efficiency or false positive probability. Still, this will need to be tested to confirm this belief.

The resulting design then has an identical interface to the Bloom filter. Firstly, indexes to each table are calculated using the cuckoo filter hashing scheme. Additionally, a fingerprint must be constructed from the input to calculate the second index and later to compare it to the memory result. This fingerprint function will merely be another hash function. These fingerprints are stored in buckets in every memory row, meaning one record can contain multiple fingerprints. After each memory returns a bucket, all fingerprints must be extracted from the buckets and compared to the constructed fingerprint, strictly following the cuckoo filter algorithm. A match is transmitted on the output if any fingerprint from the two buckets is equivalent to the input key fingerprint. The whole split cuckoo filter is illustrated in 4.3.

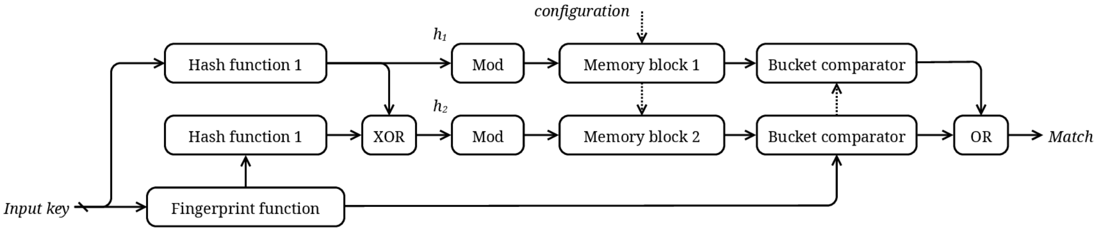


Figure 4.3: The cuckoo filter block diagram.

The cuckoo illustration also shows that the hashing scheme used in hardware is the same as discussed in theory. For comparison, the cuckoo index calculation approach is shown in (4.1).

$$h_1(x) = \text{hash}(x), \tag{4.1}$$
$$h_2(x) = h_1(x) \oplus \text{hash}(x\text{'s fingerprint}). \tag{4.2}$$

Compared to the Bloom filter, where the false positive rate depends on the number of inserted rules, the cuckoo's error rate only depends on the fingerprint size. Due to this, the hash function implemented for this purpose will need to have a configurable width.

4.3 Xor filter design

Xor probing filters are statics filters, meaning nothing can be added or removed once the filter is constructed. Moreover, because the filter's construction is only done once in its lifetime, there is little to no reason to implement the construction algorithm in the hardware as it would only waste hardware resources and be particularly tricky to implement.

If only the query for the filter needs to be implemented in hardware. Then the hardware component's design is very straightforward, containing only the query condition for the xor filter with the same interface as the Bloom and the cuckoo filter, as depicted in figure 4.4.

Similar to the designs already presented, firstly, indexes and fingerprints are calculated, then memory access is made, followed by the bitwise xor of the results and a final comparison with the input key fingerprint.

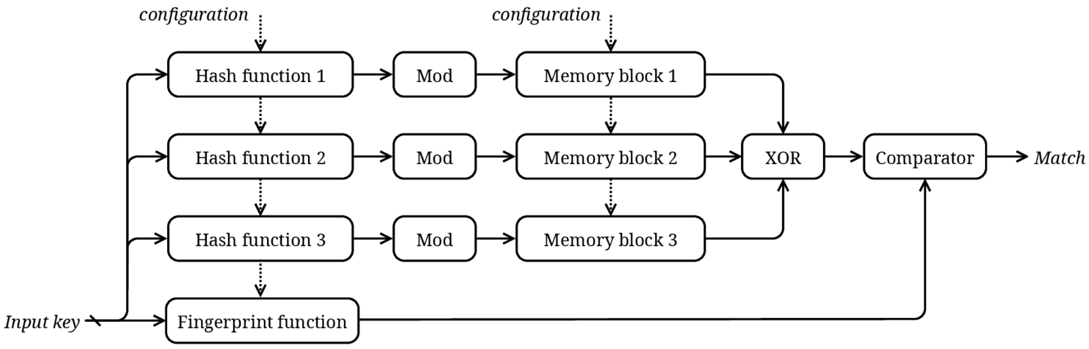


Figure 4.4: The xor filter block diagram.

The design also splits the memory array into three blocks, one for each indexing hash function. At first glance, this change is similar to the one made in the cuckoo's design. However, it is essential to note that the xor filter construction algorithm is much more complex than others. Furthermore, this change will require changes in this construction algorithm as the original xor filter uses only one array. This could affect the filter space efficiency as three arrays instead of one will need to be scaled for the filter to construct successfully. How much will the arrays need to be scaled, and how will this affect the filter construction probability will need to be tested experimentally.

Also, one can see that the design shown on 4.4 is a hardware equivalent to the xor query equation presented in the xor filter algorithm (4.3). It is important to note that array B has been split.

$$B_1[h_1(x)] \text{ xor } B_2[h_2(x)] \text{ xor } B_3[h_3(x)] = \text{fingerprint}(x). \tag{4.3}$$

Similarly to the cuckoo filter, the false positive rate of the xor filter only depends on the fingerprint's size, so the fingerprint's hash must also be width configurable. In addition, the hashes for indexing each memory array must be configurable if the xor construction algorithm fails with the current hash configuration.

4.4 Filter environment design

As three filters will be implemented, configuring and using them as one would be ideal, simplifying the construction algorithms and overall filter design. Also, there is no point in querying an uninitialised filter. A shared filter environment was designed to solve all these problems. This environment should achieve the following:

- Load keys to be inserted.
- Construct filter model in software.

- Configure the filter memories with constructed filter.
- Read filters statistics about matched and unmatched keys.
- Disable and enable the filter.

This shared design can be split into software and hardware parts, where the user interacts with the software part, which constructs and communicates with the hardware. The hardware part then configures the filter according to the software commands. This flow is illustrated in figure 4.5.

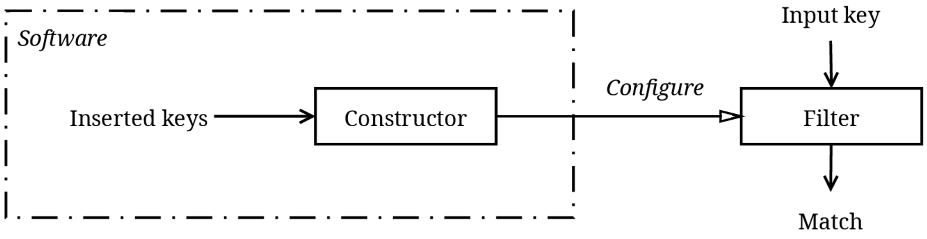


Figure 4.5: Filter environment block diagram.

To describe this environment in detail. Firstly the FPGA is configured with desired filter and specified parameters. With the prepared filter, the software configurator is presented with a ruleset. Filter parameters are read from the FPGA so that the software can construct a filter model. If the model construction is successful, the configurator will begin data transfer to set all memories in the filter required for operation. After this process, the filter is ready for use.

It is important to note that even though Bloom and cuckoo filters are dynamic and can be changed during operation, above mentioned environment does not include these features and threat all filters as static.

This environment will not be entirely identical for all filters. All filters require very different construction processes and values to be written to hardware. For example, one memory record in Bloom always has one bit. The same memory record is a product of the bucket and fingerprint sizes in cuckoo filtering. Moreover, the xor filter needs not only filter memory arrays to be transferred but also the configuration of each hash function.

The following section describes how this environment is precisely implemented and how it affects the filtering.

Chapter 5

Implementation

5.1 Filter environment implementation

Firstly, all three filters were implemented using VHDL according to the design, and their appropriate construction algorithms were written in C. The software constructor was split into two parts.

- FilterCTL: Specific filter application made by the user.
- FilterLIB: Filter construction algorithms and hardware communication backend.

As any filter can be used based on any key type, with the split implementation, it is up to the user to decide the filtering criteria.

To demonstrate the FilterLIB interface, an example implementation of FilterCTL was created. This implementation consists of parsing IPv4 and IPv6 addresses from a file and converting them to their binary form. The chosen filter type is then instantiated and configured with the parsed addresses. FilterCTL can also enable and disable filtering behaviour, read the number of matched and unmatched keys and write a ruleset id to distinguish between different configurations, fully utilising the FilterLIB interface. An example of the software flow is illustrated in figure 5.1.

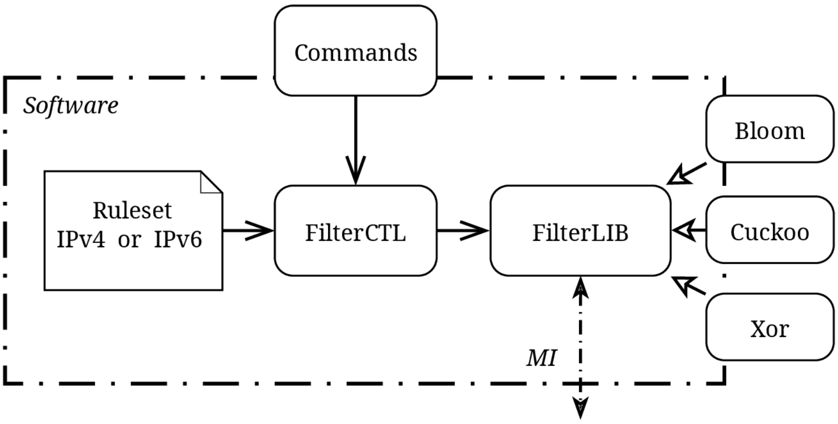


Figure 5.1: Software environment flowchart.

The FilterLIB itself has three implementations, one for each algorithm. Nonetheless, as all filters are treated as static, the interfaces are equivalent. Each implementation is internally split into two parts, construction algorithm and hardware communication backend. When a filter model is initialised in software, the communication backend finds the hardware filter and reads its configured properties as filter size, fingerprint size, bucket size, and the like. Depending on the filter type. Properties are saved into a software filter model.

The construction algorithm starts when the FilterTL sends an array containing the ruleset. The construction succeeds if the hardware filter matches the ruleset's requirements. After the software model is constructed, it should be identical to the hardware filter. This property enables the communication backend to only copy this model to the hardware. The communication with hardware is done through a PCIe bus decoded to an MI interface with the help of libraries and hardware components developed by the CESNET association [3].

This communication backend does not configure the hardware filter itself. Instead, a configurator component, shared among all three filters, receives the software requests and performs the required read or write operations. This component contains the necessary registers for the MI protocol and configures the filter by writing memory records in each table individually. The hardware environment can be seen in figure 5.2.

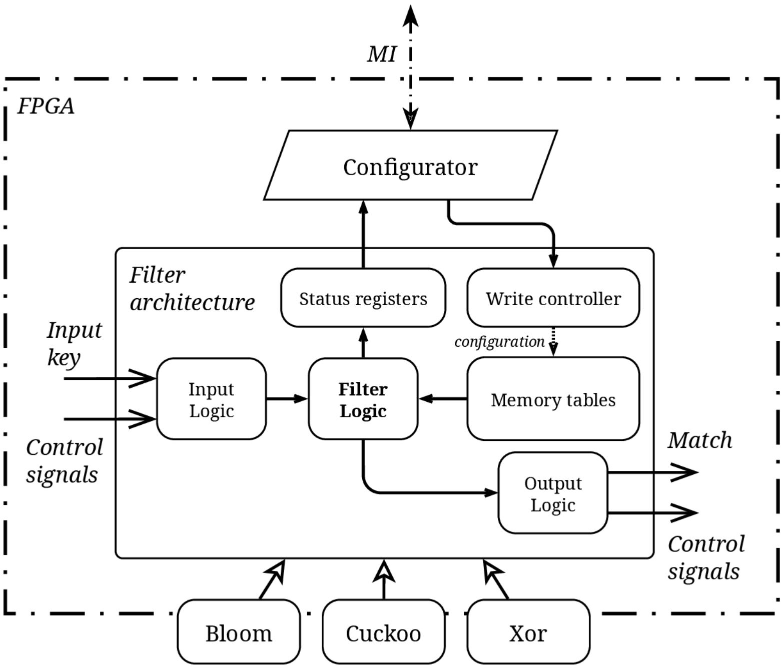


Figure 5.2: Hardware environment flowchart.

Apart from the configurator, the three filters have more in common. Mainly, write controller, status register, output and input logic. The write controller interprets the address and value from the configurator and correctly executes a write operation to a specified memory table. Status registers store the number of matched and unmatched keys. Lastly, the input and output logic mainly drives control signals like destination and source ready.

The implementation of filter logic and its construction algorithm is entirely different for each filtering method and therefore is described separately.

5.2 Bloom filter

The Bloom filter logic was implemented according to the design. However, some parts of the design were slightly changed to decrease the false positive rate and increase throughput.

The key idea is the same. Bloom has only one parameter, the number of hashes. For each hash, create a pipeline, where firstly, the input key will be bit shifted, hashed, transformed into a memory index, read from memory and lastly, all memory results will be accumulated. Creating a match output.

CRC family hashes, provided by the CESNET association [2], were used to avoid implementing a new hash function. This is ideal as the randomness of the CRC is more than enough for filtering usage, as stated in [16]. Moreover, a software implementation of these hashes was also provided, making it easy to integrate into the software constructor. This implementation is crucial as the same hash function must be used in software construction to ensure the output tables work identically.

Nevertheless, the Bloom filter performed below expectations when using only a 1-bit rotation for every hash. It was experimentally tested that at least a 4-bit rotation is needed to overcome this issue. Rotating every hash function input by 4 bits is functioning as expected. However, when many hash functions are needed, the bit shift can quickly become longer than the input key. For example, in a fifteen-hash bloom filter with 32-bit wide rules, the last rotation must be shifted by 56 bits. Conseqenting in repeating the same input and creating identical hash functions that drastically decrease the filter's performance.

The hardware implementation combats this by using two different implementations of CRC32 hash, increasing the shift for every other hash. This change results in an 18-bit shift on the last hash function in the last example, which is acceptable for the 32-bit input. Effectively a third compared to the original number of rotated bits. An example implementation of 4-hash Bloom can be seen in figure 5.3.

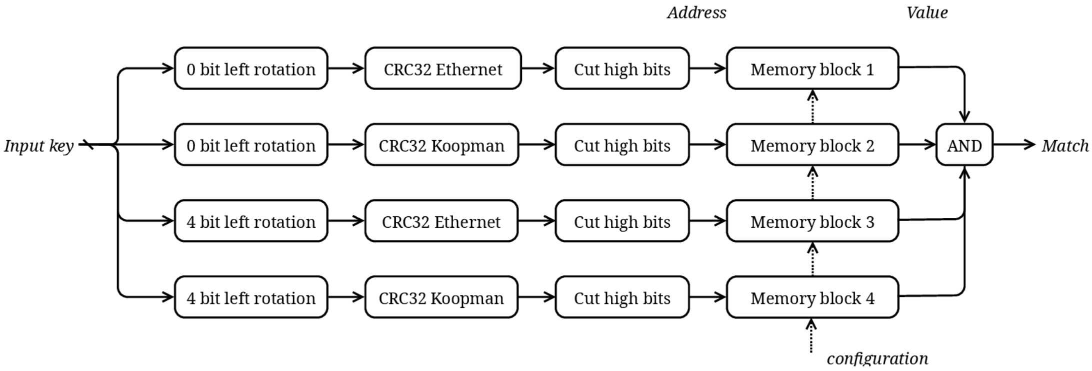


Figure 5.3: Block diagram of the Bloom filter logic.

Another change is a replacement for the modulo operation. Instead of finding the modulo of the resulting hash with the memory size, the hash is cut according to the address width of the memory. However, only powers of 2 must be allowed as the memory size to ensure this property works.

The software construction of the memory tables above is a question of many bitwise operators in C that simulate the hardware behaviour, as the read and write operations in Bloom are very similar.

5.3 Cuckoo filter

Cuckoo filter implementation uses the same hashing functions as presented before. In contrast to the Bloom filter, the cuckoo's algorithm does not require different hashes for each index, as one index is calculated via the fingerprint.

The cuckoo, however, requires another hash for the fingerprint function. The fingerprint hash is particularly important as this function's size will determine the filter's false positive rate. Five possible hash functions were implemented to achieve a broad spectrum of false positive rates—precisely, 4, 8, 16, 32 and 64-bit CRC hashes. To achieve a fingerprint length between 1 and 64-bit, the closest widest hash is first used and then cut according to memory fingerprint sizes.

What hash will be used for the fingerprint depends on the filter configuration, and only one will be generated in the final design. The whole cuckoo implementation is depicted on 5.4.

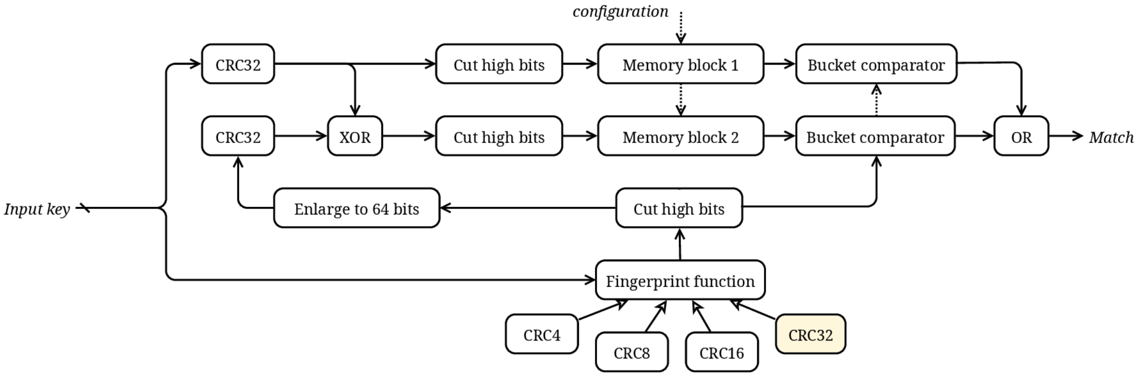


Figure 5.4: Block diagram of the cuckoo filter logic.

In order to simplify the software construction algorithm, the output fingerprint gets enlarged to 64 bits before the index hash is performed. This enlargement adds zeroes regardless of the fingerprint size, which is easily transferable to C when working with 64-bit integers. Without this, the following hash function would otherwise give different results in software and hardware.

Unlike the Bloom filter, this construction algorithm involves all problems related to cuckoo hashing. This means first filling the filter and reorganising the tables when a collision happens, as was shown in the cuckoo filter algorithm description 3.2. One important thing to realise is that no matter how the software reorganises the tables, the hardware architecture will always work as expected, as any key inserted in the table will always sit on either index i or j in their respective tables, given by the hashing scheme.

The software then fills all tables and resolves conflicts by pushing other keys into alternative locations according to xor duality property (5.1).

$$j = i \oplus \mathrm{hash}(\mathrm{fingerprint}). \tag{5.1}$$

The insertion is repeated until all keys are inserted, in which case the memory tables are sent to the hardware component, or otherwise, the filter reaches maximal key concentration, and the construction will fail. The tables are considered full if a key cannot be inserted after some predetermined amount of relocations.

5.4 Xor filter

Like the cuckoo filtering algorithm, the xor filter's false probability rate depends only on the fingerprint size. The original xor filter example implementation only used 8 or 16 bits long fingerprints. The hardware design uses the same fingerprint generation described in the cuckoo filter implementation, expanding the original approach to any fingerprint from 1 to 64 bits.

On the other hand, the indexes required for memory access are calculated similarly to the Bloom filter. However, unlike the Bloom filter, the three hashes used to calculate the memory indexes must be configurable. This reconfigurability is achieved through a seed register. The inputs are always offset by four bits, and this seed is added to each input rotator. So if the construction algorithm fails, the seed is incremented, and all three hashes are changed.

This implementation is more convenient than configuring each hash, allowing the transfer of only one value to the hardware component and thus controlling the filter, as seen in 5.5. This approach limits the possible amount of hashes. However, as the construction algorithm has a high chance of success, it should not run out of possible hashes.

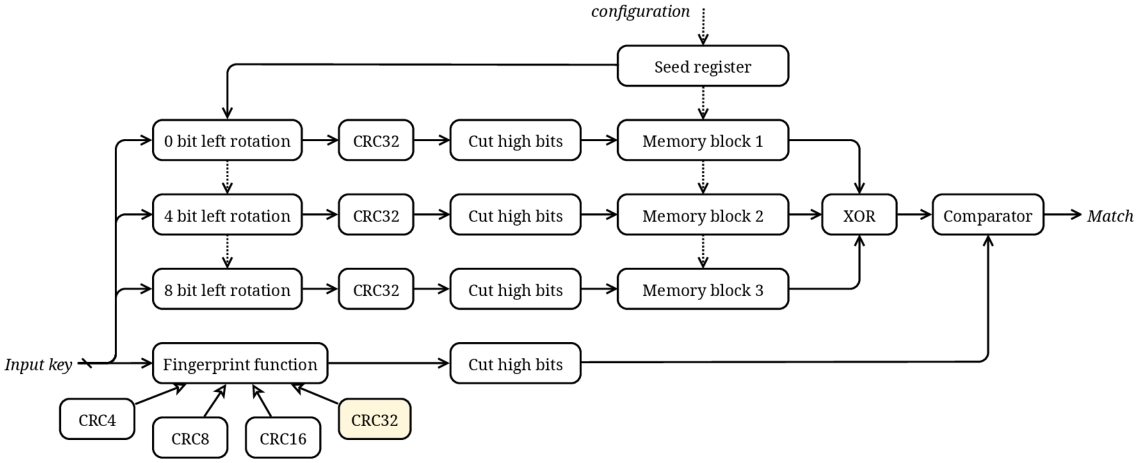


Figure 5.5: Block diagram of the xor filter logic.

Theoretically, the construction algorithm may fail many times in a row, and the seed register can shift the input out of bounds. However, the chance of this happening is improbable as the construction algorithm has a high success rate when scaled correctly. Furthermore, even if this happened, the software would consider the current ruleset too large, and the construction would fail, avoiding any unexpected states.

Moreover, to avoid adding any new functionality to the filter environment, the seed register behaves like a memory table, and from the software's point of view, the xor filter has four tables, except that the seed register only has one value. The write controller described in the filter environment implementation 5.1 ensures the seed register is appropriately used.

Construction algorithm

The biggest challenge in any xor probing algorithm is the construction. As the construction of the xor filter follows an algorithm to build acyclic 3-partite random hypergraphs [7], attempting to split the filter's memory array into multiple arrays for each hash function is no trivial task and requires substantial changes to the xor filter construction algorithm itself.

The construction algorithm contains four data structures.

- Array H: This array contains all keys after the hashes or three elements per one key. This array can contain multiple elements in one index.
- Queue Q: Queue containing elements alone in one index in the array H.
- Stack σ: Stack containing pairs with indexes in H and their values. Element ready to be inserted into the final array.
- Array B: The constructed xor filter.

To provide a concise summary of the algorithm, first define the sizes of H and B as $1.23 * n + 32$, where n represents the number of keys inserted. Next, fill H with all the keys to be inserted. Only add keys to Q if they hash to an index in H exclusively. Once Q is populated, process each key individually, removing all three hashed key variants from H.

As these elements are removed from H, new elements suitable for Q may be generated and added to Q. When an element is removed from H, consider it complete and add it to the stack σ. Continue this process until the size of σ equals the number of keys. Lastly, iterate through σ to map the values with their corresponding fingerprints in the final array B.

The new split xor construction algorithm works in the same way. However, the array H and the final array B must be split, as both arrays are accessed via the three hash functions. Both array sizes are allocated a third of the original $1.23 * n + 32$ xor filter memory budget to ensure the same space efficiency. The filter size $1.23 * n + 32$ was experimentally found out by the original xor filter authors [7]. Any size smaller than this will likely fail the construction process. The splitting is illustrated on H in 5.6.

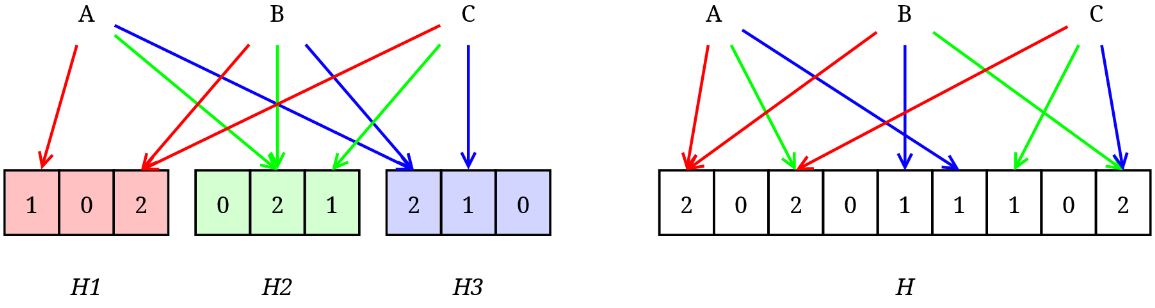


Figure 5.6: The splitting of array H.

The same hash functions are drawn with the same colour, and the number in the array indicates the number of elements in H. With this change, it is clear that the size of the filter stays the same, and each array will be accessed only using its hash function. However, this will only work assuming the number of alone elements stays the same, as the construction algorithm success rate depends on this property.

To better visualise the success rate consider the example in 5.7. Two filters with the same size and the same elements are to be constructed. However, the filter on the right hashed the elements, making the construction impossible.

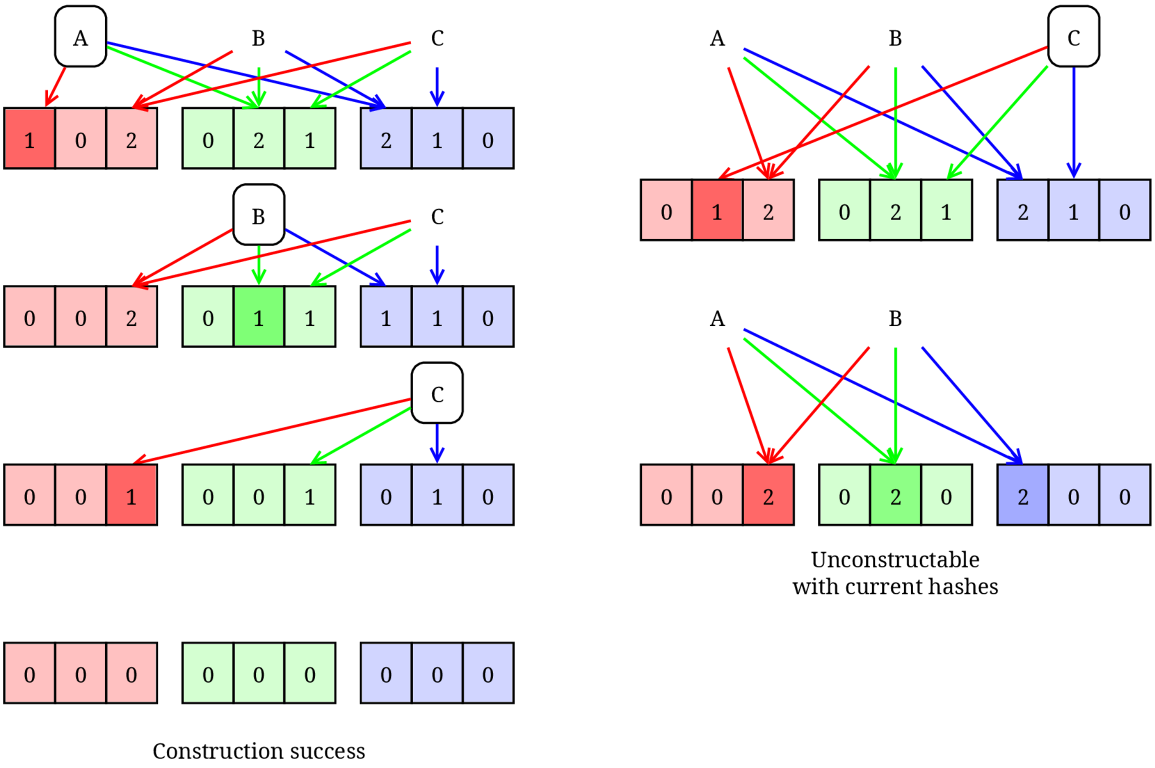


Figure 5.7: The success rate visualisation.

This inability of construction is usually solved by rehashing the elements with different hashes. However, this split algorithm could result in the construction failing most of the time, making the filter useless. A unit test for the xor library was created to see if this change influenced the success rate. All tested filters were successfully built in this testing environment, and no difference in success rate was found.

The last change in the xor construction is in the mapping step. Stack σ now contains the element and two indexes, one index i for the three arrays and the second index j for indexing in the array. The mapping step goes through this stack and evaluates the xor fingerprint for each element according to the new split xor equation (5.2).

$$B_i[j] \leftarrow \text{fingerprint}(x) \text{ xor } B_0[h_0(x)] \text{ xor } B_1[h_1(x)] \text{ xor } B_2[h_2(x)]. \tag{5.2}$$

As can be seen from the equation, every array B is now only indexed using its hash function, similar to the hardware implementation. Lastly, with all three tables filled, the split xor filter is constructed and ready to be transferred to the hardware component.

5.5 Verification

As the filtering algorithms were not just implemented but changed to improve their performance for hardware architectures, proper testing was needed to ensure that the changes did not cause any unexpected behaviour. For this reason, this work expanded on the original thesis assignment and a UVM verification environment was built to test the filters and their construction algorithms.

This UVM environment was built using components provided by the CESNET association. Thanks to this, only a few verification parts, mainly the sequences, tests, model and scoreboard, had to be written.

The verification test functions as follows. Firstly the FilterCTL program is built. A specified number of IPv4 or IPv6 rules are generated and saved to a file and the generating sequence. The filter constructor is then run with this file ruleset, and the filter is configured. The test sends the internally saved ruleset to the model. This model is the simplest filter with an array containing the ruleset and a matching function that checks if the input key is in this array. The result from the model is then compared to the response of the simulated component.

When the filter is configured and the model has the ruleset, the test sends transactions to the filter input signals. These transactions are often only a random combination of inputs created by the provided sequences. However, as a random IP address would hardly match against even a large ruleset, the test also sends transactions from a rule sequence configured with random addresses from the ruleset. This ensures that both negative and positive matches are generated. Furthermore, as the verification model is an exact matching filter, only a false negative is considered an error, and all false positives are counted to evaluate the filter's false positive rate.

The filter's status is read when all transactions are sent, containing the number of matched and unmatched transactions according to the hardware component. This is then compared to the same values in the verification to ensure all transactions were accepted correctly.

Moreover, this environment also allows one to compute the filter's size and compare it to the number of inserted keys, which can be used to calculate the space efficiency of the filter. This UVM verification was used to test all filter's false positive rates and space efficiency, making the calculations and evaluations of the results much more convenient. The actual measured results and comparisons are introduced in the next section.

Apart from the simulations, all hardware filters were synthesised and tested in a hardware application using a simple design for writing into the filter's input and reading the resulting match signal.

Chapter 6

Evaluation

In order to measure the all-around usability of mentioned methods, four evaluation criteria are used. The first two focus on the error rate and the ruleset compression efficiency, and the other pair represents the hardware requirements:

- false positive rate,
- bits per element,
- maximal frequency,
- configurable logic blocks or CLBs usage.

A false positive rate is used to measure the accuracy of the filter. This criterion is enough, as false negatives are not generated in the methods mentioned earlier. This false positive rate will decline as the filter's size enlarges with the same-sized ruleset. In other words, a filter with more bits representing one element will make fewer mistakes. This can be enumerated by the bits per element value representing the ability to compress the ruleset. The bits per element measure and the false positive rate are often used together. Luckily, both criteria can be measured using simulation and the verification environment.

However, various methods will use the hardware resources differently. Moreover, superior filters can become unpractical due to their hardware usage. Synthesis is performed on each tested design to measure the hardware occupancy of each filter. From the resulting reports, the worst negative slack, CLBs usage and the number of different RAM resources were gathered. The maximal operating frequency was then calculated using each design's known clock period and the worst negative slack.

All presented synthesis results were measured using the Xilinx Vivado 2019.1 and targeted a Virtex UltraScale+ XCVU7P device.

To make the best use of all the measured criteria. The most favourable configurations for Bloom and cuckoo filters had to be chosen to compare the filters against each other. This is important as the only shared parameter is the filter size. Other parameters are different for each method, like the number of hashes and fingerprint sizes. However, there was no need to pick any unique configurations in xor filtering, as the fingerprint size is the only parameter influencing the false positive rate and thus, no parameter needed to be fixated. How and why these configurations were picked and how much each parameter changes the filter performance is discussed below.

6.1 Bloom filter configurations

Apart from the filter size, the Bloom filter has only one parameter, the number of hashes. This parameter determines the number of pipelines in the hardware architecture, each with one hash and a memory. More pipelines result in more complex logic as more hash functions must be generated.

It is also important to note that the only parameter influencing the bits per element in the Bloom filter is the number of inserted rules. This is a speciality of the Bloom algorithm because changing the number of inserted elements in other filters does not change the false positive probability. The Bloom filter is a fingerprint-less method with a perfect 0% false positive rate when empty and steadily increasing until 100% when the whole Bloom filter is filled with ones. Therefore it is possible to insert any number of rules into any configuration. The number of hashes then only optimises this false positive rate growth for specific ranges of inserted elements.

From the figure 6.1, we can see how using a different number of hashes impacts the false positive rate.

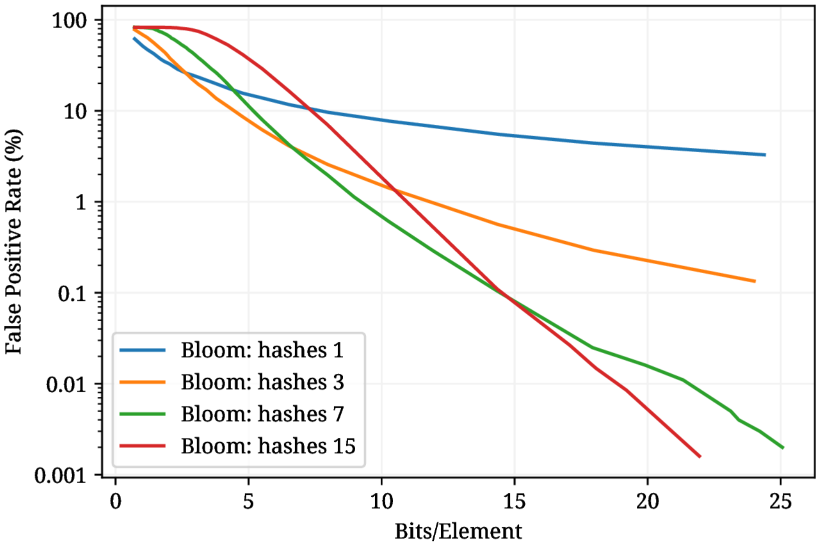


Figure 6.1: The impact of the number of hashes in the Bloom filter.

As expected, there is no best configuration. The graph confirms that increasing the number of hashes in the filter is better for more precise filtering. We can also calculate the optimal number of hashes $k_{optimal}$ with the number of inserted rules n and the filter size m using (6.1), as was discussed in the analysis of the Bloom filter 3.1.

$$k_{optimal} = \frac{m}{n} \ln 2 \approx \frac{9m}{13n} \tag{6.1}$$

The most interesting configuration seems to be the Bloom filter with seven hashes (green), as it performs the best ranging from a 2% false positive rate to around 0.1%. Moreover, it stands reasonably in a high false positive rate of around 10% and low rates of around 0.01%.

The 3-hash configuration (yellow) is the best for a high false positive rate. However, unlike the 7-hash, this configuration starts to branch off quickly.

On the other hand, when using the Bloom filter mainly for low false positive rate applications below 0.1%, the 15-hash Bloom filter (red) is compelling, clearly beating the 7-hash in this category. Therefore the 7-hash and 15-hash were chosen as the prominent Bloom filter representatives.

On top of this, thanks to the analysis of the Bloom filter in [16], it is possible to estimate the false positive rate for every configuration with the same parameters as in the optimal hash function formula using (6.2).

$$\epsilon \approx \left(1 - e^{-kn/m}\right)^k \tag{6.2}$$

Using this equation, we can confirm that the Bloom filter design performs according to expectations, as seen in figure 6.2.

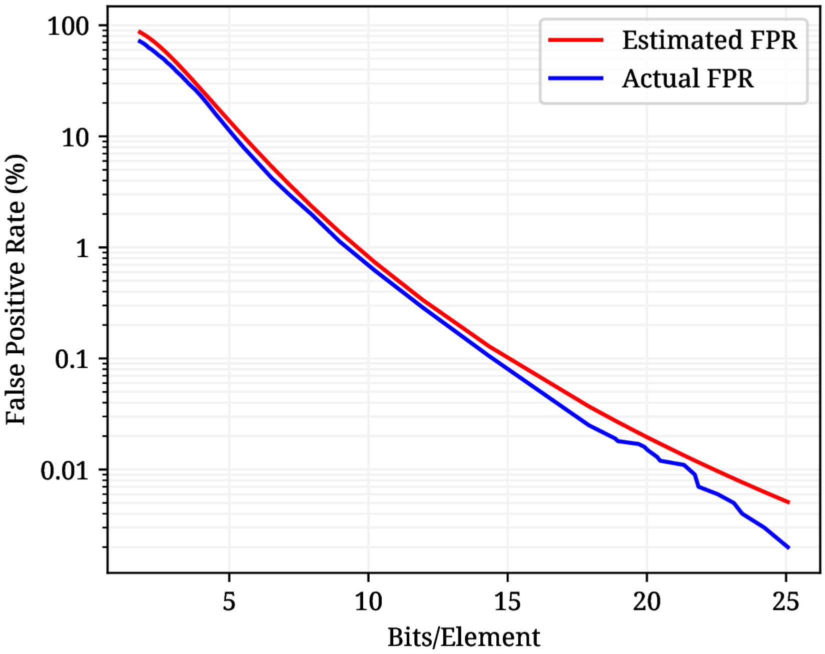


Figure 6.2: The expected and measured false positive rate of the Bloom filter.

We can see that the implemented Bloom filter's false positive rate is always below the expected rate. This is interesting as the estimation is targeted for the standard Bloom filter algorithm rather than the implemented memory-split version. We can assume from this that the implemented variant of the Bloom filter performs on par with the classic implementation.

Also, the inconsistencies in the actual false positive rate at around 20 bits per element are caused by insufficient transactions in simulation and thus can be ignored.

6.2 Cuckoo filter configurations

The cuckoo filter has two different parameters fingerprint and bucket size. Compared to the Bloom filter, the cuckoo's false positive rate can only be changed by adjusting the fingerprint size. Therefore the fingerprint size is used to modify the false positive rate. However, to only have one best configuration for the final comparison with other filters. The bucket size needed to be set to a constant, which achieves the best performance. To determine how the bucket size influences the false positive rate. Five bucket sizes were tested.

The false positive rate was gradually changed for each bucket size configuration by incrementing the fingerprint size. Each configuration was filled with the maximum number of rules to ensure that the bits per element measure was as favourable as possible, making the number of rules constant for every configuration.

Moreover, one configuration is represented as a dot in contrast to the Bloom filter, where one configuration is a line. This was because, in Bloom, changing the false positive rate is possible without changing the filter configuration. Therefore one Bloom configuration can experience any false positive rate. In contrast, cuckoo and xor filter configurations have predetermined false positive rates.

The original cuckoo filter authors recommend using 4 or 8 as the primary bucket size. However, to expand on this, 1, 2, 4, 8 and 16-sized buckets were tested, each with fingerprint size ranging from 1 to 64.

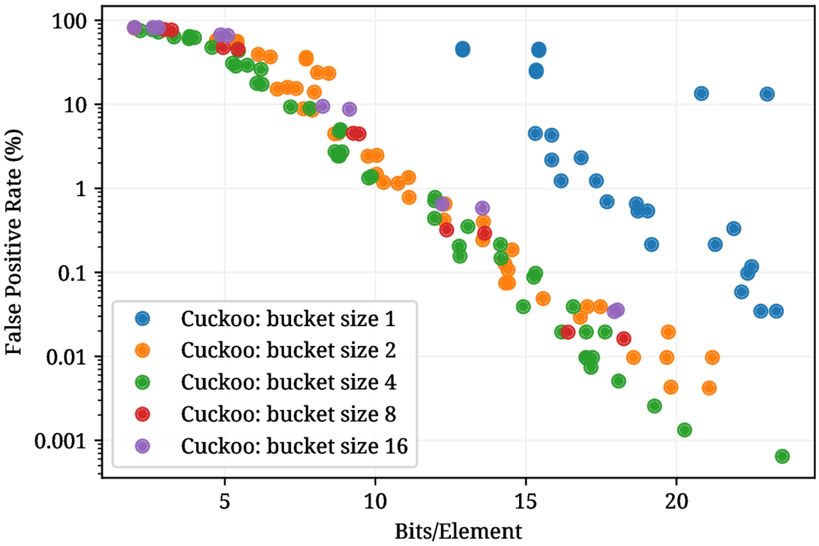


Figure 6.3: The impact of the number of buckets in the cuckoo filter.

As seen in the figure 6.3, the filter performance is similar when the number of fingerprints in the bucket is not one. This result makes sense, as with only one fingerprint in the bucket (blue). The construction algorithm is forced to make more table reconstruction. Therefore the maximum number of reconstructions is reached faster, and the filter is assumed at maximum capacity, resulting in inferior bits per element rating.

Other configurations performed better. Nonetheless, the filter with bucket size 2 (yellow) experienced a significant spread across the same false positive rate. This means that even though the configuration had a wider fingerprint and should perform better, it performed worse than expected. This is mainly visible in around 20 bits per element on the cuckoo with bucket size two.

The spreading can be caused by the fact that all simulations were run only once, and due to the cuckoo construction being heavily influenced by the random decisions in table reconstruction, some configurations can appear inferior.

Nevertheless, a configuration with four fingerprints per bucket (green) looks most efficient overall and is also recommended by the original authors as a good default and therefore was chosen for the final comparison.

6.3 Filter comparison

After the most favourable configurations for all filters were picked, we can compare their performance regarding the false positive rate and space efficiency.

To summarise, the following configurations were picked: the 7-hash and 15-hash Bloom filters, the cuckoo filter with each bucket sized for four fingerprints and the xor filter. In order to change the false positive rate, the Bloom filters were slowly filled with rules, and the cuckoo and xor filters' fingerprint sizes were changed.

The final results are shown in the figure 6.4.

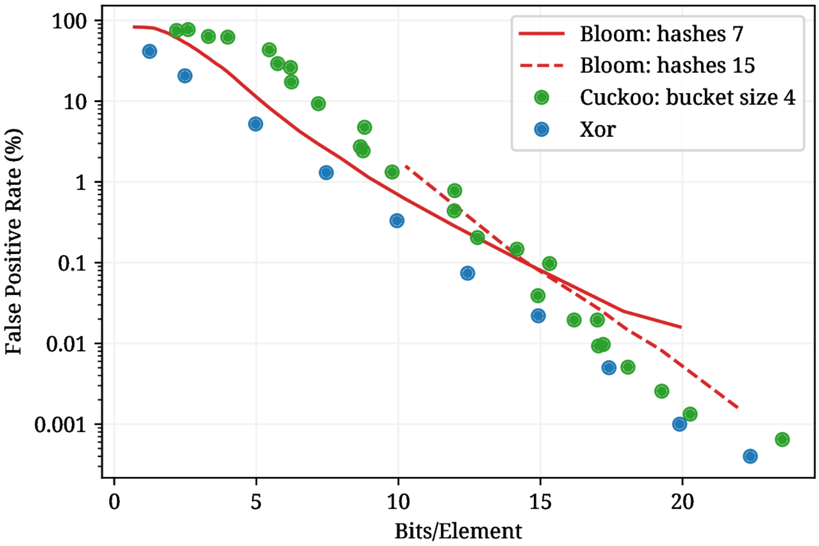


Figure 6.4: Picked configurations of all three filters compared.

Given the results, it is clear that the xor filter (blue) is superior to others in any false positive rate category. This is especially visible in the practical range of around 0.3% false positive rate, where the xor filter achieves an exceptional 10 bits per element. Apart from this, the xor filter performs very well in all categories, as was expected from a xor probing filter.

Bloom (red) performs better than the cuckoo filter (green) when the false positive rate exceeds 0.2%. Below this boundary, even the more precise 15-hash Bloom fails to match the error rate of the cuckoo filter. This switch was expected as Bloom is considered better than the cuckoo filter, only for a higher false positive rate. This switch between Bloom and cuckoo is also visible in other studies [7].

Another expected switch should happen somewhere below 0.001% false positive rate. Theoretically, the cuckoo filter should be ahead of the xor filter in this category. However, this was not measured. This is likely because the simulation result must be more precise to capture this very low false positive rate. The number of tested transactions needed to be increased to make the results more precise and suitable for higher error rates.

Nevertheless, as the number of transactions reached millions, it became almost impossible to simulate these extremely precise filters. Testing these configurations directly on hardware would be ideal, as hundreds of millions of transactions are not a problem. This problem is left open for future work.

FPGA utilisation

One configuration for cuckoo and xor filter and both 7-hash and 15-hash Blooms were picked to compare each method regarding their used resources on FPGA. Each configuration was scaled from small filter sizes to around 4000 Kb filters. To put this size into perspective. The number of inserted rules ranged from under a hundred to over 2 million. The granularity of this increase was limited by the fact that the address sizes needed to be a power of 2, as was described in the implementation. For every configuration, the hardware usage was gathered and plotted in relation to the filter size as illustrated in figure 6.5. The picked configurations were selected with around a 0.3% false probability rate. Specifically, cuckoo with bucket size four and 12-bit fingerprint, xor filter with 8-bit fingerprint and both Bloom configurations are present.

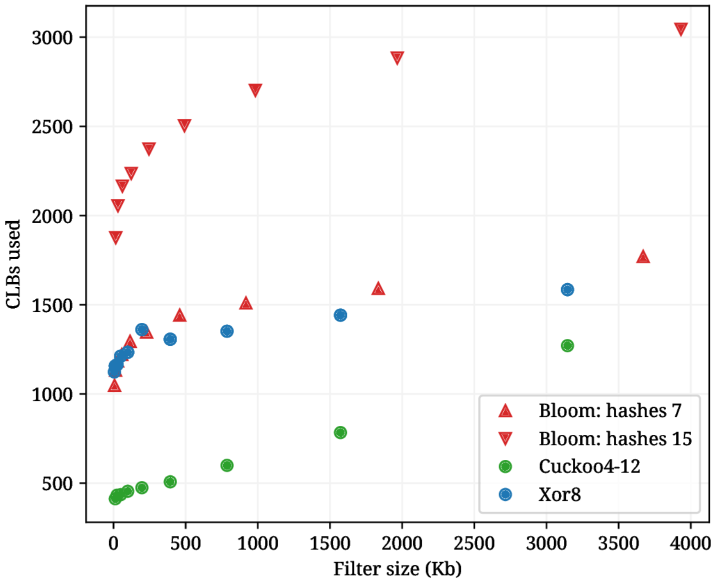


Figure 6.5: CLB usage of the picked configurations.

We can see that the cuckoo filter is the most efficient method regarding FPGA resources, especially in small filter sizes. This makes the cuckoo filter an interesting pick as it uses less than half FPGA logic compared to the xor filter and hash-7 Bloom. Moreover, compared to Bloom with 15 hashes, the cuckoo filter uses less than a third CLBs.

Regarding other filters, the xor filter is on par with hash-7 Bloom. Moreover, from the difference between the hash-15 and hash-7 Bloom configurations, we can see that additional hash functions come at a price. Precisely the hash-15 configuration is around 1.8x more expensive than then hash-7.

Filter configurations summary

The table 6.1 shows the best filter configuration for 5%, 0.3%, 0.02% and 0.005% false positive rate categories. Each configuration was picked because it had the lowest bits per element value in each category. Also, the false positive rates are not entirely the same, as guessing the suitable parameters for each filter to get the same false positive rate is almost impossible.

The names in the table indicate the configuration of each filter. The Bloom filter's name indicates the number of hashes. The number of buckets is first shown in the cuckoo filters, followed by the fingerprint size. Lastly, the xor filters' suffix also follows the fingerprint size.

Best filter configurations <30000 rules

	False positive rate	bits\element	max freq (Mhz)	CLBs
Bloom7	4.2%	6.5	**810**	1048
Cuckoo4-7	4.7%	8.8	500	**417**
Xor4	5.2%	**5**	336	693
Bloom7	0.29%	12	**810**	1048
Cuckoo4-12	0.21%	14	370	**508**
Xor8	0.33%	**10**	400	1164
Bloom7	0.027%	17	**810**	1876
Cuckoo4-15	0.02%	17	420	**523**
Xor12	0.022%	**15**	405	1213
Bloom15	0.008%	19.2	**810**	1876
Bloom7	0.005%	23.1	**810**	1048
Cuckoo4-17	0.0051%	18.1	456	**506**
Xor14	0.005%	**17.4**	362.8	1022

Table 6.1: Picked configurations overview.

From the shown configurations, we can again observe that the xor filter has the best bits per element in all categories, and the cuckoo filter is the most moderate considering its resource usage. Moreover, both Bloom configurations achieve the fastest maximal frequencies of around 800 MHz and can operate at almost double the frequency compared to other filters.

Bloom also has better bits per element in the 5% and 0.3% false positive categories than the cuckoo filter. As expected, the bits per element are more or less equal in the

0.02% category, and lastly, the cuckoo filter beats the Bloom filter in the last most precise category.

The goal was to achieve 100 Gbps and higher speeds since the filter is planned for use in CESNET projects focused on high-speed networks. According to achieved frequencies, it can handle 100 and 200 Gbps. 400 Gbps is a future work plan and can be achieved with multi-packet processing per clock cycle.

From the findings above, it is clear that the xor filter fulfilled its expectations as the most efficient filter. However, when the false positive rate is not a problem, Bloom can achieve much higher throughput than the others. Lastly, the cuckoo filter is a good alternative when FPGA resources are the primary concern.

We can also compare the probabilistic filters to their exact counterparts. It is clear that an exact matching filter, when configurated with 32-bit rules (IPv4), requires a 32-bit per element. Likewise, when 128-bit rules (IPv6) are used, 128 bits will be needed for each element. With this in mind, a xor filter with a false positive rate of around 0.3%, in the table 6.1, will need three times less memory when compared to the exact filter with 32-bit rules and less than twelve times memory when 128-bit rules are used. This difference can be enlarged further by matching multiple concatenated fields that create a new key.

Chapter 7

Conclusion

This thesis discussed the theory and design of probabilistic filters for hardware applications. Three methods were presented: the Bloom filter, cuckoo filter, and xor probing filter. These algorithms are used to efficiently store and query large sets of data in a space and time-efficient manner, with the trade-off of a certain probability of false positives.

The basic principles of each algorithm and their advantages and limitations were described. Furthermore, hardware design changes to each filter were presented to make them more viable for hardware implementation. The most changed method was the xor filter, where the fingerprint size was generalised, and the construction algorithm was changed to split the memory into three independent tables. Additionally, all designs were verified using the developed verification environment.

The final results revealed that the xor filter outperformed the other filters regarding space efficiency and false positive rate. In the practical range of around 0.3% false positive rate, the xor filter achieved an exceptional 10 bits per element. The Bloom filter was more efficient than the cuckoo filter when the false positive rate exceeded 0.2%, while the cuckoo filter was the most resource-efficient on FPGA, especially for smaller filter sizes. These results and the whole work were also presented at the Excel@FIT 2023 student conference.

In terms of future work, several improvements can be made to enhance the performance of these filters. Implementing multi-packet processing per clock cycle will allow for 400 Gbps processing. Making Bloom and cuckoo filters dynamic will enable adding and removing rules. Other potential enhancements include implementing counting Bloom filters to support deletion, adding more tables to the cuckoo filter to improve the false positive rate, and updating the xor construction to the latest binary fuse filter to increase space efficiency and reduce the false positive rate.

In summary, probabilistic filters are a powerful tool for various applications in computer networks, and their hardware design can improve their performance and efficiency. However, it is crucial to carefully evaluate the trade-offs between accuracy and efficiency and choose the most appropriate algorithm for a specific application.

Bibliography

[1] Bloom, B. H. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. jul 1970, vol. 13, no. 7, p. 422–426. DOI: 10.1145/362686.362692. ISSN 0001-0782. Available at: https://doi.org/10.1145/362686.362692.

[2] CESNET. *CESNET/ndk-mod-internal: Private repository with IPs/modules developed internally and typically further licensed as non-exclusive.* Available at: https://gitlab.liberouter.org/ndk/ndk-mod-internal/-/tree/main/base/ff.

[3] CESNET. *CESNET/ndk-sw: Linux driver and SW tools for Network Development Kit (NDK).* Available at: https://github.com/CESNET/ndk-sw.

[4] Dillinger, P. C. and Walzer, S. *Ribbon filter: practically smaller than Bloom and Xor*. arXiv, 2021. DOI: 10.48550/ARXIV.2103.02515. Available at: https://arxiv.org/abs/2103.02515.

[5] Dixit, M., Barbadekar, B. V. and Barbadekar, A. B. Packet classification algorithms. In: *2009 IEEE International Symposium on Industrial Electronics*. 2009, p. 1407–1412. DOI: 10.1109/ISIE.2009.5215939.

[6] Fan, B., Andersen, D. G., Kaminsky, M. and Mitzenmacher, M. D. Cuckoo Filter: Practically Better Than Bloom. In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. New York, NY, USA: Association for Computing Machinery, 2014, p. 75–88. CoNEXT '14. DOI: 10.1145/2674005.2674994. ISBN 9781450332798. Available at: https://doi.org/10.1145/2674005.2674994.

[7] Graf, T. M. and Lemire, D. Xor Filters: Faster and Smaller Than Bloom and Cuckoo Filters. *ACM J. Exp. Algorithmics*. New York, NY, USA: Association for Computing Machinery. mar 2020, vol. 25. DOI: 10.1145/3376122. ISSN 1084-6654. Available at: https://doi.org/10.1145/3376122.

[8] Graf, T. M. and Lemire, D. Binary Fuse Filters: Fast and Smaller Than Xor Filters. *ACM J. Exp. Algorithmics*. New York, NY, USA: Association for Computing Machinery. mar 2022, vol. 27. DOI: 10.1145/3510449. ISSN 1084-6654. Available at: https://doi.org/10.1145/3510449.

[9] Guinde, N., Ziavras, S. and Rojas Cessa, R. Efficient packet classification on FPGAs also targeting at manageable memory consumption. In:. January 2011, p. 1 – 10. DOI: 10.1109/ICSPCS.2010.5709753.

[10] Gupta, P. and McKeown, N. Algorithms for packet classification. *IEEE Network*. 2001, vol. 15, no. 2, p. 24–32. DOI: 10.1109/65.912717.

[11] Matoušek, P. *Síťové aplikace a jejich architektura*. Brno: VUTIUM, 2014. ISBN 978-80-2143-766-1.

[12] Mitzenmacher, M. and Broder, A. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*. january 2004, vol. 1, no. 4. DOI: 10.1080/15427951.2004.10129096.

[13] Pagh, R. and Rodler, F. F. Cuckoo hashing. *Journal of Algorithms*. 2004, vol. 51, no. 2, p. 122–144. DOI: https://doi.org/10.1016/j.jalgor.2003.12.002. ISSN 0196-6774. Available at: https://www.sciencedirect.com/science/article/pii/S0196677403001925.

[14] Qi, Y., Xu, L., Yang, B., Xue, Y. and Li, J. Packet Classification Algorithms: From Theory to Practice. In: *IEEE INFOCOM 2009*. 2009, p. 648–656. DOI: 10.1109/INFCOM.2009.5061972.

[15] Sateesan, A., Vliegen, J., Daemen, J. and Mentens, N. Hardware-oriented optimization of Bloom filter algorithms and architectures for ultra-high-speed lookups in network applications. *Microprocessors and Microsystems*. 2022, vol. 93, p. 104619. DOI: https://doi.org/10.1016/j.micpro.2022.104619. ISSN 0141-9331. Available at: https://www.sciencedirect.com/science/article/pii/S0141933122001582.

[16] Tarkoma, S., Rothenberg, C. E. and Lagerspetz, E. Theory and Practice of Bloom Filters for Distributed Systems. *IEEE Communications Surveys and Tutorials*. 2012, vol. 14, no. 1, p. 131–155. DOI: 10.1109/SURV.2011.031611.00024.

[17] Wada, T., Matsumura, N., Yasudo, R., Nakano, K. and Ito, Y. Efficient implementations of Bloom filter using block RAMs and DSP slices on the FPGA. *Concurrency and Computation: Practice and Experience*. 2021, vol. 33, no. 12, p. e5475. DOI: https://doi.org/10.1002/cpe.5475. e5475 cpe.5475. Available at: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5475.