Macrothink Institute™

Network Protocols and Algorithms
ISSN 1943-3581
2015, Vol.7, No.1

# A Bloom Filter with the Integrated Hash Table Using an Additional Hashing Function

Reza Pourian

Department of Computer Engineering, Faculty of Engineering, Kermanshah Branch,

Islamic Azad University, Kermanshah, Iran

Tel: +98-9183304235 E-mail: r.e.pourian@iauksh.ac.ir

Mahmood Ahmadi

Department of Computer Engineering, Faculty of Engineering, University of Razi,

Kermanshah, Iran

Tel: +98-8334274595 E-mail: m.ahmadi@razi.ac.ir

Received: August 29, 2014 Accepted: January 3, 2015 Published: April 30, 2015

DOI:10.5296/npa.v7i1.6240 URL: http://dx.doi.org/10.5296/npa.v7i1.6240

**Abstract**

A Bloom filter is a simple space-efficient randomized data structure for representing a set in order to support membership queries. In recent years, Bloom filters have increased in popularity in database and networking applications. A Bloom filter has two steps that called programming and membership query. In this paper, we introduce a new approach to integrate a hash table (HT) with Bloom filter to decrease the HT access time. This means that when a Bloom filter for an incoming item is programmed, the incoming item simultaneously is stored in a HT. In addition in the membership query step, if the query is successful, simultaneously the address of item in the HT is generated. Furthermore, we analyze the average bucket size, maximum search length and number of collisions for the proposed approach and compare to the fast hash table (FHT) approach. We implemented our approach in a software packet classifier based on tuple space search with the $H3$ class of universal hashing functions. Our results show that our approach is able to reduce the average bucket size, maximum search length and number of collisions when compared to a FHT.

***Keywords:*** Bloom filter, network processor, packet classification, universal hashing.

## 1 Introduction

Bloom filters are frequently utilized in databases and networking applications, such as distributed databases, packet classification, packet inspection, forwarding, IP route lookup,

and distributed web caching [1][2][3][4][5][6]. The Bloom filter is utilized in four types of network-related applications:

1. Collaboration in overlay and peer-to-peer networks: Bloom filter is utilized in overlay and p2p networks.
2. Resource routing: Bloom filters allow probabilistic algorithms for locating resources.
3. Packet routing and classification: Bloom filters can be used to speed up packet routing protocols.
4. Measurement: Bloom filters provide a useful tool for measurement infrastructures used to create data summaries in routers and other network devices.

Most network devices, e.g., routers and firewalls, require the processing of incoming packets (e.g., classification and forwarding) at wire speeds. These devices mostly incorporate special network processors that are comprised of a processor core with several memory interfaces and special co-processors that are optimized for packet processing [7]. The well-known gap between processor and memory performance has been a major source of concern for computing in general; this problem is exacerbated in packet processing systems. Such memory bottlenecks can be overcome by the following mechanisms: hiding of memory latencies through parallel processing and reducing the memory latencies by introducing special memory architectures [8]. An approach to achieve higher lookup performance is to utilize the Bloom filter data structure that recently has been introduced utilized in embedded memory for network processors [9][10]. Indeed, due to its high applicability in network packet processing, some modern network processors incorporate dedicated Bloom filter units in their implementation [9].

Each Bloom filter has two steps includes programming and querying. In the programming stage, given a string X, which is a member of the signature set, a Bloom filter computes k many hash values on the input X by using k different hash functions. Then it uses these hash values as index to the m-bit long lookup vector. It sets the bits corresponding to the index given by the hash values computed. It repeats this procedure for each member of the signature set. In the query stage, for an input string Y, Bloom filter computes k many hash values by utilizing the same hash functions used in programming of the Bloom filter. Bloom filter looks up the bit values located on the offsets (computed hash values) on the bit vector, and, If it finds any bit unset at those addresses, it declares the input string to be a non-member of the signature set, which is called a mismatch. Otherwise, it finds all the bits are set, it concludes that input string may be a member of the signature set with a certain probability (false positive probability), which is called a match.

A Bloom filter version that is often utilized in network processors and packet processing applications is the counting Bloom filter. In the Bloom filters, for an input item, a separated hashing operation through the hash table (HT) is needed. This means that after the membership checking to access the memory a separated hashing function and HT access is required. To decrease the memory access time in the Bloom filters, a caching policy and fast hash table (FHT) techniques can be used, respectively [2][9]. The mentioned caching policy exploits the Bloom filter properties to decrease the number of memory accesses [11][2]. The FHT technique reduces the memory access time but has some limitations as: high processing time

due to incremental update, reconsideration of all items when inserting items, and limited use as it works only in conjunction with counting Bloom filter and thereby reducing its applicability for other applications. In this paper, we propose an approach to integrate the access of the HT with Bloom filter operations that called Bloom Filter integrated with HT using an Additional Hashing Function (BFAHF). In the other hands, in the programming step of the Bloom filter a HT address to store an item using an additional hashing function is generated. In query step the address of the stored item for the possible access using the mentioned additional hashing function is generated. In addition, our approach operates independently from the counters in the counting Bloom filter used in FHT and bit-array in the standard Bloom filter. This means it can be applied with the all Bloom filter types. In this approach, we utilize an additional hashing function to select one of the generated addresses by the $k$ hashing functions in Bloom filters. The utilization of an additional hashing function has the following advantages: decreasing memory access time, and its simplicity to implement in hardware. The main contributions of our work are the following:

- Proposal of a decreasing memory access time technique for Bloom filters using an additional hashing function.
- Analysis of the proposed approach in terms of different performance metrics (e.g., average bucket size, maximum search length, and the number of collisions) compared to the FHT technique.
- Integration of the Bloom filter operation to the HT that holds the items.
- Implementation of the Bloom filter with the integrated HT using a hash-based software packet classifier.

This paper is organized as follows. Section 2 presents related work. Section 3 describes the concept of a Bloom filter. Section 4 describes the concept and architecture of Bloom filter with the integrated HT using additional hashing function. Section 5 presents our software implementation and results. In Section 6, we draw the overall conclusions.

## 2 Related Work

In this section, we take a brief look at previous works regarding the Bloom filter and its memory organization. In [3][9], an extended version of the Bloom filter is considered. They presented a FHT architecture and lookup algorithm that converts a Bloom filter into a counting Bloom filter and an associated hash bucket. The FHT improves the performance over a standard HT by reducing the number of memory accesses needed for the most time-consuming lookups. It only works in conjunction with counting Bloom filters and needs to reconsider all of the already inserted items for each item that consequently leads to longer processing time. In [10], a hash architecture called a Multi-predicate Bloom-filtered Hash Table (MBHT) using parallel Bloom filters is presented. It is generated off-chip memory addresses in the base-$2^x$ number system, $x \in \{1, 2, ...\}$, which removes the overhead of pointers. Using a larger base of number system, an MBHT reduces on-chip memory size. In [3][12], an approach to packet classification which combines architectural and algorithmic techniques is presented. The starting point is the well-known crossproduct algorithm which is fast but has significant memory overhead due to additional rules needed to represent the crossproducts. The proposed approach

modifies the crossproduct method to reduce the memory requirement. Unnecessary accesses to the off-chip memory are avoid by filtering them through on-chip Bloom filters. Compressed Bloom filters (CBF) were introduced in [13], which improved performance when the Bloom filter is passed as a message in distributed web caching protocols. The author investigated compressed Bloom filter for distributed proxy web caching and showed that by using compressed Bloom filters, proxies can reduce the number of bits broadcast, the false positive rate, and the amount of computation per lookup. The CBF is useful when a query is preformed in distributed environment that is sent a Bloom filter in the compressed form to the corresponding side.

In [2], a cache design based on the standard Bloom filter was investigated and was extended to support ageing (adding the ability to evict stale entries from the cache), bound misclassification rates, and use multiple binary predicates. It examined the exact relationship between the size and dimension of the number of flows that can be supported and the misclassification probability incurred. Additionally, it presented extensions for gracefully ageing the cache over time to minimize misclassification.

In [14], utilization of a Matrix Bloom filter in a plagiarism checking system is proposed. In this system, for each document in the library, a matrix Bloom filter is constructed then the achieved Bloom filter for queried document is compared to the set of Bloom filters. cache. In [15], the Bloom-1 and Bloom-g idea is proposed that performs membership check in one memory access, rather than $k$ memory accesses in Bloom filter, but it needs more parameters than Bloom filter and more complex operations. In the FHT [3][12], the searching and insertion criteria are performed based on the value of counters that for each operation $k$ (number of hashing functions) counters should be inspected. In our previous work [16], the output addresses of $k$ hashing functions is selected using an additional hashing function that is performed independently from the counters or bit-array in counting or standard Bloom filters, respectively. In this work, we extend the BFAH and analytically show different performance criteria such as average bucket size, maximum search length and the number of collisions. We also implemented a hash-based software packet classifier using BFAH and compare the results to other Bloom filter implementations e.g., FHT.

## 3 Bloom Filters

A Bloom filter [17] is a simple space-efficient randomized data structure for representing a set in order to support membership queries. A set $S$ $(x_1, x_2, ..., x_n)$ of $n$ items is represented by an array $V$ of $m$ bits that are initially all set to 0. A set of $k$ independent hash functions $h_1$, $h_2$, ..., $h_k$ (each with an output range between 1 and $m$) is utilized to set $k$ bits in array $V$ at positions $h_1(x)$, $h_2(x)$, ..., $h_k(x)$ for all $x$ in set $S$. More precisely, for each item $x \in S$, the bits at positions $h_i(x)$ are set to 1 for $1 \leq i \leq k$. Moreover, a location can be set to 1 multiple times. To verify whether an item $y$ is a member of the set S, the same set of hash functions is utilized to determine $h_i(y)$ (for $1 < i < k$) indicating the locations in array V to be checked whether their content is a 1. If one of these location yields a 0, $y$ is certainly not a member of the set S. If all locations yield a 1, there is a high probability that $y$ is a member of the set $S$ (*positive*). However, as increasingly more bits in array V are set to 1, one can imagine that the probability of a false positive will increase. It must be clear now that there is an inverse relation between the number of bits in the array and the false positive rate. In the extreme case, when all bits in the array are set, every search will yield a (false) positive [17][18][9]. The false positive probability is given as follows:

$$P = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1 - e^{\frac{-kn}{m}}\right)^{k} \quad (1)$$

In this equation, $n$ represents the number of items, $m$ represents the number of bits in the bit-array and $k$ represents the number of hashing functions. For a given $m$ and $n$, the value of $k$ (the number of hash functions) that minimizes the probability is as follows:

$$k = \frac{m}{n} ln2 \approx \frac{9m}{13n} \approx 0.7\frac{m}{n} \quad (2)$$

## 4 Bloom Filter with the Integrated Hash Table using an Additional Hashing Function

In this section, we present our concept and analysis of Bloom filter with the integrated HT using an additional hashing function.

### *4.1 The Bloom Filter with the integrated hash table using an Additional Hashing Function (BFAHF) architecture and concept*

The Bloom filter is space-efficient membership query tool that is used in network processing applications. In packet processing application that utilized the Bloom filter after the programming and querying steps of Bloom filter it needs to access the sought item. The item is accessed independently from the Bloom filter operations. Therefore, this question must be addressed. Is it possible to perform the Bloom filter operations and HT access simultaneously that can save the memory access time? The answer is possible by integrating Bloom filter operation and HT accesses. The integration is performed using an additional hashing function. The proposed solution has the following features:

- Decreasing of memory access time, by overlapping the Bloom filter operations and accessing HT.
- It can be applied to the all Bloom filter types. This is because, it works independently from bit-array of Bloom filter.
- In the FHT the searching for an incoming item, requires inspection of $k$ counters in counting Bloom filter while in our approach a single hash value is computed.

The architecture of the Bloom filter with the integrated HT using an additional hashing function is depicted in Figure 1.

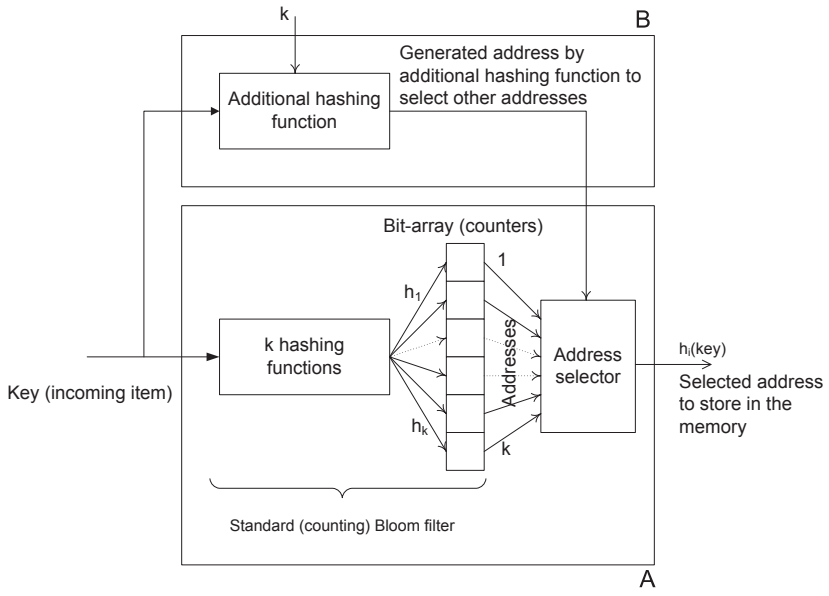


Figure 1: A Bloom filter architecture with the integrated HT using an additional hashing function.

In Figure 1, an incoming item is hashed by $k$ hashing functions and the corresponding bits are set (or counters are incremented in the counting Bloom filter) (see block A in Figure 1). Subsequently, one of the generated addresses by the $k$ hashing functions is selected by another hashing function. This address is used to store the item in question. The additional hashing function receives the incoming key and selects one of $k$ received addresses to store in the memory. The output of the additional hashing function is a number that represents the index of one of $k$ addresses pointed to by hashing functions (see block B in Figure 1). The generated addresses by the $k$ hashing functions (block A in Figure 1) and generated number by the additional hashing function is processed by the address selector unit in block A. This component selects an address among the incoming addresses. It should be noted, the additional hashing function works alongside the others $k$ hashing functions. Therefore, the selected address by the additional hashing function and generated addresses by $k$ hashing functions are available at the same time in the address selector unit. Additionally, in the FHT, the counters should be searched to find the minimum counter but in this approach only one hashing key by the additional hashing function is calculated.

Figure 2 depicts an example of how the integrated HT using an additional hashing function operates using the Bloom filter. In this example, we utilize the divide hashing function "$key \ mod \ k$" as additional hashing function where $k$ represents the number of hashing functions and index of incoming items is supposed as input key. This is due to the simplicity of this hashing function in the example.

Figure 2 (A) depicts a Bloom filter before any insertion therefore the values of the counters or bits in the bit-array are zero. In the first step, rule $R0$ is hashed to addresses 0, 2, and 4 using $h0$, $h1$, and $h2$ hashing functions (see Figure 2 (B)). The value of the counters are incremented (or bit-array is set) and then a generated addresses by hashing functions should be selected to store the rule $R0$ in the hashed address. Therefore, an additional hashing function selects one address out of $k$ generated addresses. The additional hashing function selects address $0$ that

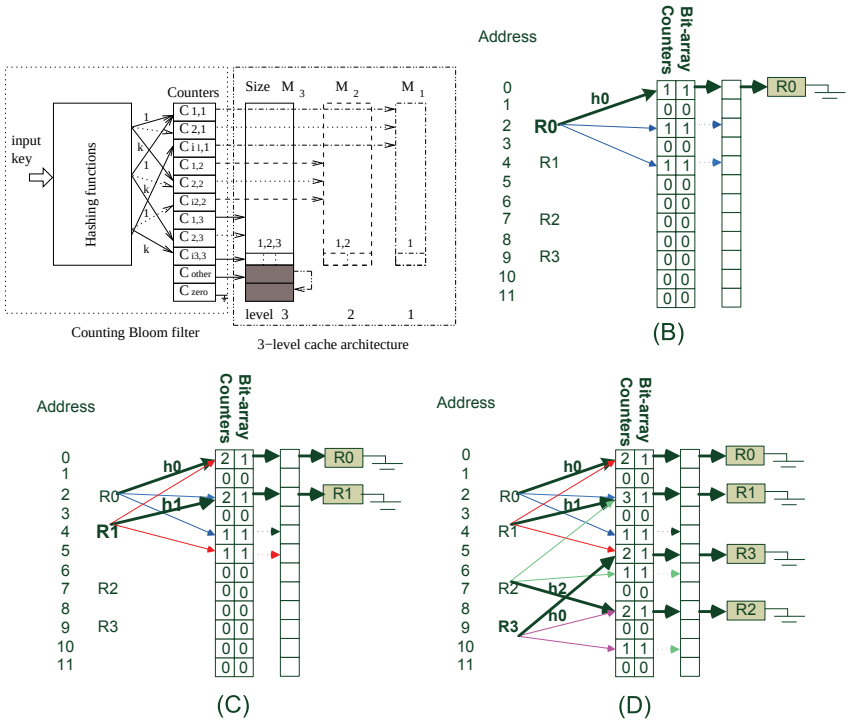


Figure 2: A Bloom filter with the integrated HT using an additional hashing function. (A) An empty Bloom filter (B) The Bloom filter after the insertion of rule $R0$ (C) The Bloom filter after the insertion of rules $R0$ and $R1$ (D) Final Bloom filter after the insertion of four rules.

was generated by hashing function $h0$ and $R0$ is stored in the address $0$. This is because the index of $R0$ that is '0', is divided to 3 ($0 \ mod \ 3 = 0$) and, finally address $0$ is selected to store the rule $R0$. In this example number of hashing function is 3.

Subsequently, $R1$ is hashed to addresses $0$, $2$, and $5$ (see Figure 2 (C)). The values of counters are incremented (bit-array is set) to 2, 2, and 1, respectively. Using the additional hashing function "$key \ mod \ k$", address $2$ that was generated by hash function $h1$ is selected to store the $R1$. As depicted Figure 2 (C), $R2$ is hashed to addresses $2$, $6$, and $8$ and address $8$ that generated by $h2$ is selected to store rule $R2$. Finally, rule $R3$ after the hashing to addresses $5$, $8$ and $10$ is stored at address $5$.

### *4.2 Analysis of the Bloom Filter with the Integrated Hash Table using an Additional Hashing Function*

We analyze the FHT and the Bloom filter with the integrated HT using an additional hashing function. In this analysis, the average bucket size, the maximum search length and the number of collisions are investigated.

#### 4.2.1 Analysis of average bucket size

The average bucket size is defined as the total number of stored items in the Bloom filter divided by the number of non-empty entries (buckets). In the Bloom filter with $n$ items and $nk$ hashing operations, the probability that a bucket receives exactly $j$ insertions is expressed as:

$$p\left(b(i)=j\right)=\left(\begin{array}{c} nk \\ j \end{array}\right)\left(\frac{1}{m}\right)^{j}\left(1-\frac{1}{m}\right)^{nk-j} \tag{3}$$

In Eq. 3, $b(i)$ represents the number of items in $bucket(i)$. In this equation, to calculate the probability of an empty bucket, $j$ is set to 0. Therefore, the probability of non-empty buckets is equal to 1 minus the probability of the empty buckets. Consequently, the probability of a non-empty bucket is calculated as follows:

$$p(non-empty\,buckets)=\left(1-\left(1-\frac{1}{m}\right)^{nk}\right) \tag{4}$$

From Eq. 4, the expected number of the non-empty buckets is calculated as follows:

$$E(Number\,of\,non-empty\,buckets)=m\left(1-\left(1-\frac{1}{m}\right)^{nk}\right) \tag{5}$$

In a Bloom filter, the average bucket size is as follows:

$$Average\,bucket\,size\,in\,BF=\frac{Number\,of\,stored\,items}{Number\,of\,non-empty\,buckets}=\frac{nk}{m\left(1-\left(1-\frac{1}{m}\right)^{nk}\right)} \tag{6}$$

In the Bloom filter with the integrated HT, in first step, for $n$ items $nk$ hashing operations are performed using $k$ hashing functions and the related bits in the bit-array are set. In a Bloom filter with $m$ bits (bit-array size), $n$ items and $k$ hashing functions, $1/m$ represents the probability any one of the $m$ bits set by a single hashing function operating on a single input item. $(1-1/m)$ is the probability that the bit is unset after a single hash value computation with a single item. The probability that a bit is still unset after all the items are hashed into the Bloom filter by using $k$ independent hashing functions is $\left(1-\frac{1}{m}\right)^{kn}$. Therefore, the probability of set bits in the bit-array is $\left(1-\left(1-\frac{1}{m}\right)^{nk}\right)$. Consequently, the expected number of set bits in the bit-array is calculated using $t=m\left(1-\left(1-\frac{1}{m}\right)^{nk}\right)$. In the next step, an additional hashing function selects $n$ out of $t$ addresses that set in the previous step. Using Eq. 5, the number of selected items by the additional hashing function occupy $t\left(1-\left(1-\frac{1}{t}\right)^{n}\right)$ addresses, where $t$ is the number of set bits in the bit-array. Therefore, the average bucket size in the Bloom filter with the integrated HT using additional hashing function is calculated as follows:

$$Aver.\,bucket\,size\,in\,Bloom\,filter\,with\,the\,integrated\,HT=\frac{n}{t\left(1-\left(1-\frac{1}{t}\right)^{n}\right)}\;with\;t=m\left(1-\left(1-\frac{1}{m}\right)^{nk}\right) \tag{7}$$

It should be noted, in the Bloom filter with the integrated hash table using additional hashing function (BFAHF) $n$ items are stored.

FHT can be assumed as a standard Bloom filter with one hashing function that selects $n$ out of $nk$ hashed items, therefore, the average bucket size for fast HT is computed using Eq. 7 when $k=1$. Eq. 8, shows the average bucket size for the FHT.

$$Average\,bucket\,size\,in\,FHT=\frac{n}{m\left(1-\left(1-\frac{1}{m}\right)^{n}\right)} \tag{8}$$

The average bucket size for the Bloom filter with the integrated HT using additional hashing function, and FHT using Eqs 7 and 8 for the configuration $k = ln(2)m/n$ that generates minimum false positive probability is depicted in Figure 3.

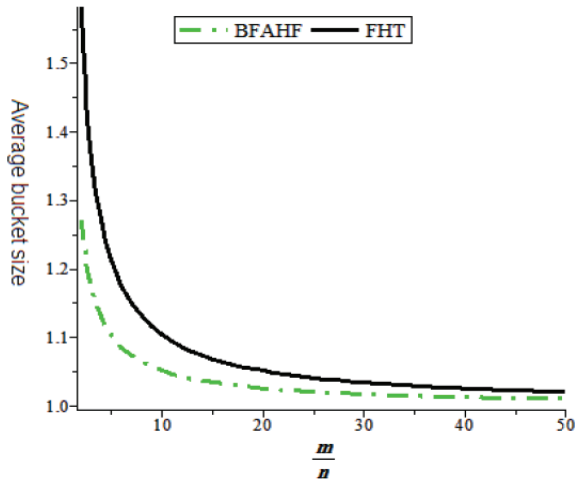


Figure 3: Average bucket size for the Bloom filter with the integrated hash table (BFAHF) and FHT when $k = \frac{m}{n}ln(2)$).

Figure 3 depicts average bucket size in term of $m/n$ ($m$ is number of bits in the bit-array and $n$ is number of the items). To generate minimum false positive probability $k = ln(2)m/n$. From Figure 3, we can observe that the shortest average bucket size belongs to the BFAHF. The graph in Figure 3 shows that Bloom filter with the integrated HT has average bucket size shorter than FHT. Additionally, from Figure 3, we can observe that when the value of $m/n$ is increased the average bucket size of Bloom filter with the integrated HT and FHT is closed to each other. In this case, for $m/n = 50$ the difference of average bucket size between Bloom filter with the integrated HT and FHT is less than 1%.

### 4.2.2 Maximum search length

The maximum search length is defined as a maximum number of items that are inserted in the buckets. It can be used as a worst case search to access an item. In the standard Bloom filter, the expected number of items which their buckets included $j$ items is equal to average number of buckets multiplied by the item per bucket (3), therefore, it is calculated as follows:

$$E(Number\ of\ items\ which\ their\ buckets\ included\ j\ items) = nkj\left(\begin{pmatrix} nk \\ j \end{pmatrix}\left(\frac{1}{m}\right)^{j}\left(1-\frac{1}{m}\right)^{nk-j}\right) \quad (9)$$

In Eq. 9, $nk$ is the number of hashed items and $j$ is the bucket size (number of items per bucket). In Bloom filter with the integrated HT, in first step, all $nk$ hashed items are hashed using $k$ hashing functions and related bits in the bit-array are set. The number of set bits in the bit-array is calculated using $t = m\left(1-\left(1-\frac{1}{m}\right)^{nk}\right)$. In the next step, an additional hashing function selects $n$ out of $t$ addresses that set in the previous step. Therefore, in a Bloom filter with the integrated HT, the probability of a bucket with $j$ items is represented as follows:

$$p\,(Bucket\ with\ j\ items) = \left( \begin{array}{c} n \\ j \end{array} \right) \left(\tfrac{1}{t}\right)^{j} \left(1 - \tfrac{1}{t}\right)^{n-j} \tag{10}$$

Using Eq. 10, the number of items with bucket size $j$ in the Bloom filter with the integrated HT is calculated as follows:

$$E(Number\ of\ items\ which\ \ bucket\ size = j) = nj \left( \left( \begin{array}{c} n \\ j \end{array} \right) \left(\tfrac{1}{t}\right)^{j} \left(1 - \tfrac{1}{t}\right)^{n-j} \right) \tag{11}$$

For FHT, maximum search length is computed when in the standard Bloom filter the number of hashing functions is set to 1. Hence, the number of items with bucket size $j$ is calculated as follows:

$$E(Number\ of\ items\ which\ \ bucket\ size = j) = nj \left( \left( \begin{array}{c} n \\ j \end{array} \right) \left(\tfrac{1}{m}\right)^{j} \left(1 - \tfrac{1}{m}\right)^{n-j} \right) \tag{12}$$

Using Eqs. 11 and 12, the maximum search length for FHT and Bloom filter with the integrated HT when $k = ln(2)m/n$ is depicted in Figure 4.

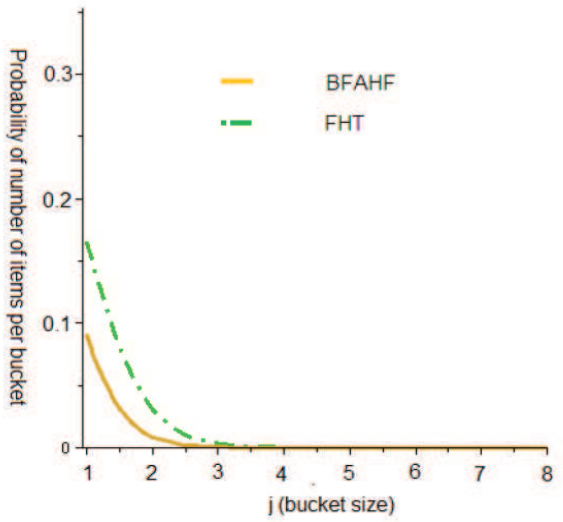


Figure 4: Maximum search length for Bloom filter with the integrated HT and FHT when $k = ln(2)m/n$.

From Figure 4, we can observe that, the maximum search length of Bloom filter with the integrated HT is longer than FHT. As an example, for $m = 10000$, $n = 2000$ and $k = 4$, 3, and 1 buckets received more than 3 items in the Bloom filter with the integrated HT, and FHT, respectively.

### 4.2.3 Number of collisions

A common problem in using hashing is collision that means the mapping of incoming items to the same HT location. Consequently, when an incoming item is hashed to a HT entry containing multiple items it must be matched to all these items resulting in a much longer processing time. In the Bloom filter, for each item, collision is detected when the number of

hashed items to the related location is larger than bucket size. Therefore, in the Bloom filter using Eq. 3, the probability of number of collisions for $i^{th}$ address in the bit-array is as follows:

$$p\left(col(i)\ with\ bucket\ size\ p\right) = \sum_{j=p}^{p_{max}-1} (j+1-p) \left( \begin{array}{c} nk \\ j+1 \end{array} \right) \left(\frac{1}{m}\right)^{j+1} \left(1-\frac{1}{m}\right)^{nk-j-1} \quad (13)$$

In Eq. 13, $col(i)$ shows the number of collisions for $i^{th}$ address when the bucket size is set to $p$, while $p_{max}$ shows the maximum bucket size. Using Eq. 13, the expected number of collisions in the standard Bloom filter is as follows.

$$E\left(Number\ of\ collisions\ in\ the\ standard\ BF\right) = nk \sum_{j=p}^{p_{max}-1} (j+1-p) \left( \begin{array}{c} nk \\ j+1 \end{array} \right) \left(\frac{1}{m}\right)^{j+1} \left(1-\frac{1}{m}\right)^{nk-j-1} \quad (14)$$

It is obvious that to calculate the number of hashed items to each address the value of $p$ is set to 1. As we discussed previously, we can extend Eq. 12 for BFAHF and FHT. In the BFAH, in first step, all $nk$ hashed items are hashed using $k$ hashing functions and related bits in the bit-array are set. The number of set bits in the bit-array is calculated using $t = m\left(1-\left(1-\frac{1}{m}\right)^{nk}\right)$. In the next step, an additional hashing function selects $n$ out of $t$ addresses that set in the previous step. Therefore, in a BFAH, using Eq. 12, the expected number of collisions is presented as follows.

$$E\left(Number\ of\ collisions\ BFAHF\right) = n \sum_{j=p}^{p_{max}-1} (j+1-p) \left( \begin{array}{c} n \\ j+1 \end{array} \right) \left(\frac{1}{t}\right)^{j+1} \left(1-\frac{1}{t}\right)^{n-j-1} \quad (15)$$

For FHT, number of collisions is computed when in the standard Bloom filter the number of hashing functions is set to 1. Hence, the number of collisions is calculated as follows.

$$E\left(Number\ of\ collisions\ FHT\right) = n \sum_{j=p}^{p_{max}-1} (j+1-p) \left( \begin{array}{c} n \\ j+1 \end{array} \right) \left(\frac{1}{m}\right)^{j+1} \left(1-\frac{1}{m}\right)^{n-j-1} \quad (16)$$

Using Eq.s 14, 15 and 16, the number of collisions for the standard, FHT and the BFAHF when $k = ln(2)m/n$ is depicted in Figure 6. In this figure, the number of collisions in the BFAHF and FHT is normalized to $n$ and in the standard Bloom filter is normalized to $nk$. In the BFAHF and FHT number of the stored items is $n$ while in the standard Bloom filter is $nk$. In addition, the value of $p_{max}$ is set to 16. This is because, the analysis by Fan, et al. [6] for counting Bloom filter shows 4-bit counter is enough for most applications. In the other words, in all types of Bloom filters the probability for each address to receive more than 16 items is very small.

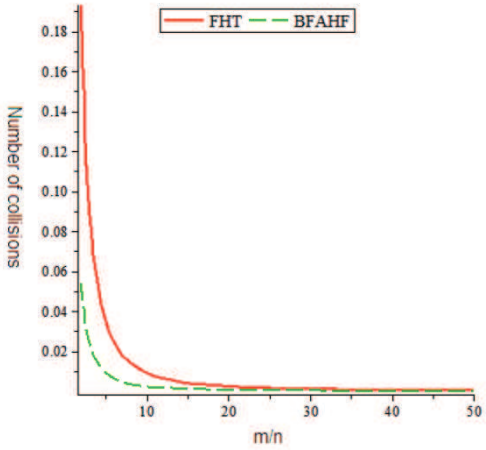


Figure 5: Number of collisions for standard, BFAHF and FHT when $k = ln(2)m/n$.

In Figure 6, we can observe that the number of collisions in BFAHF and FHT are converged when the value of $\frac{m}{n}$ is increased. As an example, for $m = 20000$, $n = 1000$ and $k = 14$ the number of collisions are 1, 4 and 2509 for FHT, BFAHF and standard Bloom filter. It should be noted that in this example, 1000 items are stored in the BFAHF and FHT and 14000 items are stored in the standard Bloom filter.

## 5 Implementation and Results

In this section, we present the implementation results of a software packet classifier that utilizes the standard, FHT and a BFAHF in tuple space packet classification and subsequently show the results.

### *5.1 Implementation*

We implemented a software packet classifier [19] based on the tuple space search that utilizes of standard Bloom filter, FHT, and BFAHF. A high-level approach for multiple field search employs tuple spaces with a tuple representing information in each packet header field specified by the rules. Srinivasan, et. al., [20][21][22] introduced the tuple space approach and the collection of tuple search algorithms. The tuple space framework utilizes of the concept of searching on prefix lengths to cope with multidimensional packet classification. In multi-dimensional packet classification, each rule defines prefix specifications on multiple packet header fields, and therefore each rule has more than one prefix length. The vector of prefix lengths of a rule is called a tuple, and the tuple space is the set of distinct tuples in a rule-set. The rules mapped to the same tuple can be searched using one hash operation. For each tuple, we utilize a Bloom filter (Bloom filter with an additional hashing function) with a set of universal hashing functions. The class of universal hashing functions is called $H3$ hashing functions that we utilize in the software packet classifier [23][24]. Based on the tuple space representation for rule-set databases and IP packets, the size of an input key is 88 bits ( 32 bit source IP address, 32 bit destination IP address, 8 bit Range-ID, 8 bit Nesting-Level and 8 bit protocol bit). The concept of Range-ID and Nesting-Level is defined to represent the tuple value of port ranges [20][21]. In this implementation, based on our observations of rule

distribution in the tuples the maximum size of tuple or address space is assumed $2^{16}$ rules for 16 bit address. Therefore, $Q_{88\times16}$ denotes a set of matrices to define an $H3$ hashing function for tuple space packet classification algorithm [25].

### *5.2 Results*

For the testing purpose, we use different rule-set databases and packet traces that were utilized by the Applied Research Laboratory in Washington University in St. Louis [26][27]. The specification of the rule-set databases and packet traces is presented in Table 1.

| **Rule database** | **FW1-100** | **FW1-1k** | **FW1-5k** | **FW1-10k** | **FW1** | **ACL1** | **IPC1** |
|---|---|---|---|---|---|---|---|
| **Number of rules** | 92 | 971 | 4653 | 9311 | 266 | 752 | 1550 |
| **Number of tuples** | 26 | 42 | 52 | 57 | 36 | 44 | 179 |
| **Packet trace** | **FW1-100** | **FW1-1k** | **FW1-5k** | **FW1-10k** | **FW1** | **ACL1** | **IPC1** |
| **Number of packets** | 920 | 8050 | 46700 | 93250 | 2830 | 8140 | 17020 |

Table 1: Rule-set databases and packet traces specification.

Table 1 includes seven rule-sets databases and packet traces. The rule-sets FW1, ACL1, IPC1 were extracted from real rule-sets and the other were generated by the Classbench benchmark. More details on Classbench, rule-set databases, and packet traces can be found in [26]. In the figures that will be presented later the horizontal axis represents two sequences of data. The sequence with label $k$ shows the number of hashing functions and later one shows the corresponding value of $m/n$ that is calculated based on Eq. 2 to minimize the false positive probability. In addition, the graphs show the average bucket size, the maximum search length and the number of collisions for all of the rule-set databases in Table 1.

#### 5.2.1 Investigation average bucket size

In Section 4.2.1, we analyzed the average bucket size and observed that the average bucket size in the BFAHF is shorter than standard Bloom filter. In here, we present the average bucket size for the standard Bloom filter, FHT and BFAHF using a software packet classifier. The graph of average bucket size for standard Bloom filter, FHT and BFAHF is depicted in Figure 6.

In Figure 6, the vertical axis represents the average bucket size in terms of the items per bucket. From this figure, we can observe that the BFAHF has the average bucket size shorter FHT. For instance, when $m/n = 26$ and $k = 18$, the difference is 2%. Furthermore, the increase in the number of hashing functions and the corresponding $m/n$ value decrease the average bucket size in BFAHF and FHT. It should be noted, when $k = 1$, two mentioned Bloom filters operate as a simple hashing system.

#### 5.2.2 Investigation maximum search length

In Section 4.2.2, we analyzed the maximum search length for FHT and BFAHF. In this section, we present the maximum search length of the FHT and BFAHF using a software packet classifier. The graph of maximum search length for the 3 Bloom filters is depicted in Figure 7.

From Figure 7, we can observe that the maximum search length for BFAHF, and FHT is almost the same. In FHT, the maximum search length is changed between 3 and 4 while for

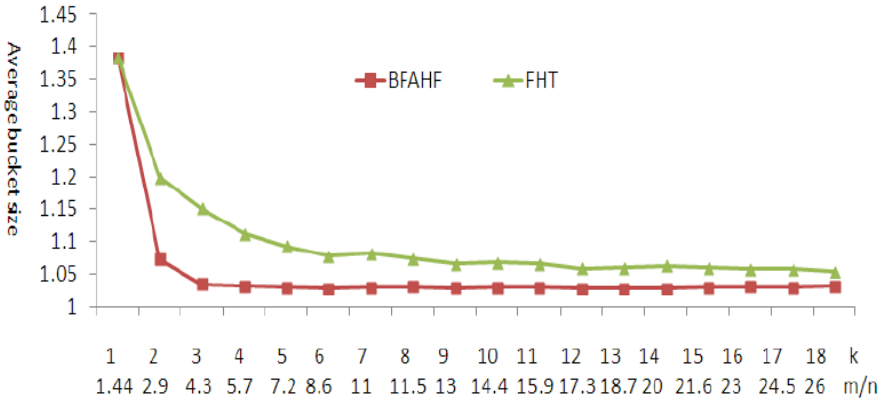


Figure 6: Average bucket size for standard Bloom filter, FHT filter and BFAHF.

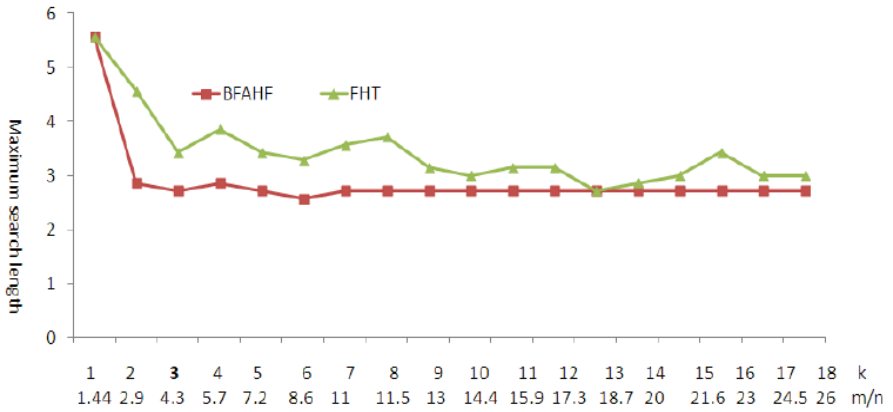


Figure 7: Maximum search length for BFAHF and FHT.

the BFAHF the maximum search length is 3. We can observe that when the $m/n$ is increased maximum search length for BFAHF and FHT converge to the same value.

### 5.2.3 Investigation number of collisions

The graph of the number of collisions for BFAHF and FHT is depicted in Figure 8. In this figure, the average number of collisions for all rule-set databases is normalized to $n$ (number of rules in rule-set database) for the FHT and BFAHF.

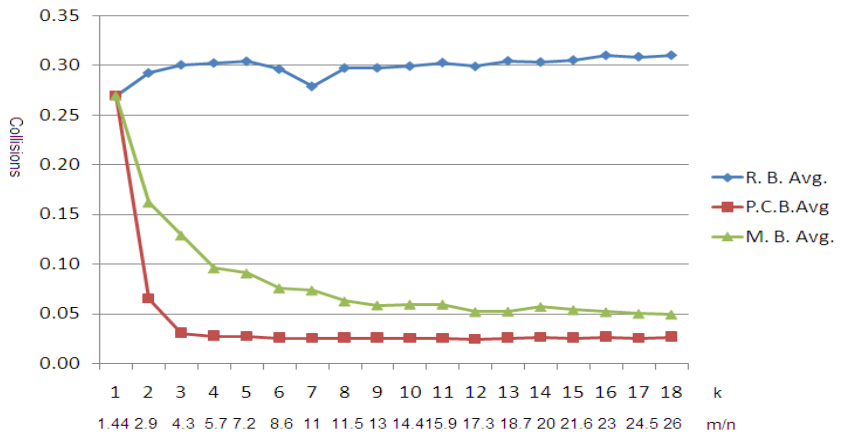


Figure 8: Average number of collisions for all rule-set databases that normalized to $n$ (number of rules in rule-set database) for FHT and BFAHF.

Based on Figure 8, we can observe that the number of collisions for FHT converges to the number of collisions in the BFAHF when the value of $m/n$ is increased.

### 5.2.4 Discussion

In general, we observed that the utilization of BFAHF decreases the average bucket size, maximum search length and the number of collisions in comparison to the FHT. As we expected, the analytical results is verified by the software packet classifier results. In addition, in the FHT, for each incoming item, $k$ counters should be investigated but in the BFAHF the address is directly selected. Furthermore, BFAHF can be applied to all Bloom filters types while the FHT only works with counting Bloom filter.

The presented results show that the BFAHF approach enhances different performance metrics compared to the FHT that this means the memory bottleneck in high performance network processing applications can be overcome.

## 6 Overall Conclusions

In this paper, we presented a new approach to decrease the memory utilization in the standard Bloom filter. We utilize an additional hashing function to select a generated address by hashing functions in the Bloom filter. Utilization of an additional hashing function increases the performance of Bloom filter (in term of average bucket size, maximum search length and number of collisions) in comparison to the FHT. Our analysis and software implementation results validate this. The main advantage of the BFAHF technique is that it be applied to all Bloom filter types but the FHT technique only works with counting Bloom filter. The FHT technique needs a parallel search among the counters pointed to by hashing functions to find the counter with minimum value while the BFAHF approach accesses the item in question only using one hash operation. We expect this approach to be useful in the design of high performance memory architectures and processing engines (e.g., forwarding, packet inspection and classification) utilized in the network processors and others network processing applications. In our future work, we use the quotient filter in the routing table design and test it using open source router environments.

## 7 Acknowledgement

The authors wish to thank the Islamic Azad University for supporting projects. This research was supported by Islamic Azad University, Kermanshah Branch, Kermanshah, Iran.

[1] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," in *Proc. 14'th Annual Allerton Conf. on Communication, Control, and Computing*, 2002, pp. 636–646. [Online]. Available: http://www.eecs.harvard.edu/~michaelm/NEWWORK/postscripts/BloomFilterSurvey.pdf

[2] F. W.-c. F. Chang and L. Kang, "Approximate Caches for Packet Classification," in *23'th Annual Conf. of the IEEE, Infocom*, 2004, pp. 2196–2207. [Online]. Available: http://dx.doi.org/10.1109/INFCOM.2004.1354643

[3] J. T. S. Dharmapurikar, S. Song and J. Lockwood, "Fast Packet Classification Using Bloom Filters," Department of Computer Science And Engineering, Washington University in St. Louis, Tech. Rep. 27, 2006. [Online]. Available: http://dx.doi.org/10.1145/1185347.1185356

[4] B. B. B. Donnet and T. Friedman, "Retouched Bloom Filters: Allowing Networked Applications to Flexibly Trade Off False Positives Against False Negatives," in *Proc. of the ACM CoNEXT 2006 conference*, 2006, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1145/1368436.1368454

[5] L. F. V. C. Esteve and M. F. Magalh, "Towards a new generation of information-oriented internetworking architectures," in *Proc. of the 2008 ACM CoNEXT Conference*, 2008, pp. 1–6. [Online]. Available: http://dx.doi.org/10.1145/1544012.1544077

[6] J. A. L. Fan, P. Cao and A. Z. Broder, "Summary Cache: A Scalable Wide-Area (WEB) Cache Sharing Protocol," *IEEE/ACM Trans. on Networking*, vol. 8, no. 3, pp. 281–293, 2000. [Online]. Available: http://dx.doi.org/10.1109/90.851975

[7] M. Ahmadi and S. Wong, "Network Processors: Challenges and Trends," in *Proc. of the 17th Annual Workshop on Circuits, Systems and Signal Processing, ProRisc 2006*, pp. 223–232. [Online]. Available: http://ce.et.tudelft.nl/publicationfiles/1221_758_Pro_Risk_Paper.pdf

[8] H. V. J. Mudigonda and R. Yavatkar, "Overcoming the Memory Wall in Packet Processing: Hammers or Ladders?" in *Proc. of Symp. on Architecture for Networking and Communications systems (ANCS-05)*. ACM Press, 2005, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1145/1095890.1095892

[9] S. D. H. Song, J. Turner and J. Lockwood, "Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing," in *Proc. of Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2005, pp. 181–192. [Online]. Available: http://dx.doi.org/10.1145/1090191.1080114

[10] H. Yu and R. N. Mahapatra, "A Memory-Efficient Hashing by Multi-Predicate Bloom Filters for Packet Classification," in *INFOCOM*, 2008, pp. 1795–1803. [Online]. Available: http://dx.doi.org/10.1109/INFOCOM.2008.242

[11] M. Ahmadi and S. Wong, "A Cache Architecture for Counting Bloom Filters," in *Proc. of 15th IEEE Int. Conf. on Networks (ICON2007)*, pp. 218–213. [Online]. Available: http://dx.doi.org/10.1109/ICON.2007.4444089

[12] J. T. S. Dharmapurikar, H. Song and J. Lockwood, "Fast Packet Classification Using Bloom Filters," in *Proc. of the ACM/IEEE Symp. on Architecture for Networking and Communications Systems*. ACM, 2006, pp. 61–70.

[13] M. Mitzenmacher, "Compressed Bloom Filters," in *Proc. of the 20th annual ACM Symp. on Principles of Distributed Computing (PODC 01)*, 2001, pp. 144–150. [Online]. Available: http://dx.doi.org/10.1109/TNET.2002.803864

[14] S. Geravand and M. Ahmadi, "An efficient and scalable plagiarism checking system using bloom filters," *Computers & Electrical Engineering*, vol. 40, no. 6, pp. 1789–1800, 2014. [Online]. Available: http://dx.doi.org/10.1016/j.compeleceng.2014.06.003

[15] T. L. Y. Qiao and S. Chen, "Fast Bloom Filters and Their Generalization," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 1, pp. 93–103, Jan 2014. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2013.46

[16] M. Ahmadi and S. Wong, "A Memory-optimized Bloom Filter using An Additional Hashing Function," in *Proc. of IEEE Globecom 2008 Next Generation Networks, Protocols, and Services Symposium*, 2008, pp. 1–5. [Online]. Available: http://dx.doi.org/10.1109/GLOCOM.2008.ECP.476

[17] B. H. Bloom, "Space /Time Trade-offs in Hash Coding with Allowable Errors," *Communication of the ACM*, vol. 13, no. 7, pp. 422–426, 1970. [Online]. Available: http://dx.doi.org/10.1145/362686.362692

[18] S. Kumar and P. Crowley, "Segmented Hash: An Efficient Hash Table Implementation for High Performance Networking Subsystems," in *Proc. Symp. on Architecture for Networking and Communications Systems (ANCS05)*, 2005, pp. 91–103. [Online]. Available: http://dx.doi.org/10.1145/1095890.1095904

[19] M. Ahmadi and S. Wong, "Modified Collision Packet Classification Using Counting Bloom Filter in Tuple Space," in *Proc. of the 25'th IASTED Int. Conf. on Parallel and Distributed Computing and Networks (PDCN 2007)*, pp. 70–76. [Online]. Available: http://ce.et.tudelft.nl/publicationfiles/1272_664_iasted_152_ahmadi.pdf

[20] V. Srinivasan, "A Packet Classification and Filter Management System," in *Proc. Int. IEEE Conf. INFOCOM*, 2001, pp. 1464–1473. [Online]. Available: http://dx.doi.org/10.1109/INFCOM.2001.916642

[21] S. S. V. Srinivasan and G. Varghese, "Packet Classification Using Tuple Space Search," in *Proc. Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1999, pp. 135–146. [Online]. Available: http://dx.doi.org/10.1145/316194.316216

[22] D. E. Taylor, "Models, Algorithms, and Architectures for Scalable Packet Classification," Ph.D. dissertation, Department of Computer Science and Engineering Washington

University, 2004. [Online]. Available: http://www.arl.wustl.edu/~jst/studentTheses/dTaylor-2004.pdf

[23] J. C. Lawrence and M. N. Wegman, "Universal Classes of Hash Functions," in *Proc. of the 9th annual ACM symposium on Theory of computing*. ACM Press, 1977, pp. 106–112. [Online]. Available: http://dx.doi.org/10.1145/800105.803400

[24] E. F. M. V. Ramakrishna and F. Bahcekapili, "Efficient Hardware Hashing Functions for High Performance Computers," *IEEE Trans. Computer.*, vol. 46, no. 12, pp. 1378–1381, 1997. [Online]. Available: http://dx.doi.org/10.1109/12.641938

[25] M. Ahmadi and W. S, "Hashing Functions Performance in Packet Classification," in *Proc. of Int. Conf. on the Latest Advances in Networks (ICLAN-2007)*, 2007, pp. 127–132. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.62.9642

[26] H. Song, "Evaluation of Packet Classification Algorithms," 2006, http://www.arl.wustl.edu/ hs1/PClassEval.html. [Online]. Available: http://www.arl.wustl.edu/~hs1/PClassEval.html

[27] D. E. Taylor, "Evaluation of Packet Classification Algorithms," http://www.arl.wustl.edu/ hs1/PClassEval.html3.FilterSets, 2006, applied Research Laboratory Department of Computer Science and Engineering Washington University in Saint Louis.