A Cache Architecture for Counting Bloom Filters

Mahmood Ahmadi and Stephan Wong
Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
{mahmadi, stephan}@ce.et.tudelft.nl

Abstract— **Within packet processing systems, lengthy memory accesses greatly reduce performance. To overcome this limitation, network processors utilize many different techniques, e.g., utilizing multi-level memory hierarchies, special hardware architectures, and hardware threading. In this paper, we introduce a multi-level memory hierarchy and a special hardware cache architecture for counting Bloom filters that is utilized by network processors and packet processing applications such as packet classification and distributed web caching systems. Based on the value of the counters in the counting Bloom filter, a multi-level cache architecture called the cache counting Bloom filter (CCBF) is presented and analyzed. The results show that the proposed cache architecture decreases the number of memory accesses by at least 51.3% when compared to a standard Bloom filter.**

Keywords: Cached counting Bloom filter, network processor, packet processing, cache level.

I. INTRODUCTION

Most network devices, e.g., routers and firewalls, need to process incoming packets (e.g., classification and forwarding) at wire speeds. These devices mostly incorporate special network processors that are comprised of a programmable processor core with several memory interfaces and special co-processors that are optimized for packet processing[1]. However, the performance of these network processors is usually hampered by slow memory accesses. Such memory bottlenecks can be overcome by the following mechanisms: hiding of memory latencies through parallel processing and reducing the memory latencies by introducing a multi-level memory hierarchy incorporating special-purpose caches [9]. A poorly designed cache memory can critically affect the performance of network processor since the number of memory accesses required for each lookup can vary. Therefore, high-throughput applications requires search techniques with more predictable worst-case lookup performance. One such approach to achieve higher lookup performance is to utilize the Bloom filter data structure that recently is utilized in embedded memories.

A Bloom filter is frequently utilized in network processing (areas), such as packet classification, packet inspection, forwarding, p2p networks, and distributed web caching [4][5][6][10]. In this paper, we present the counting Bloom filter and analyze the value of counters in the counting Bloom filters. Afterwards, we introduce a new cache architecture called the *cached counting Bloom filter* (CCBF). In addition, the pruning procedure to optimize the memory utilization in the counting Bloom filter is described. Based on the counting Bloom filter analysis, we propose two multi-level cache architectures and, subsequently, present the performance analysis. The performance metric is the number of accesses in different cache levels of the CCBF compared to those on the regular Bloom filter. The results show that the number of accesses is decreased by at least 51.3% when utilizing a 3-level cache architecture. In this 3-level cache, we further determine a balance between the different levels to arrive at approximately equal sizes for the levels. This paper is organized as follows. Section II presents related work. Section III describes a counting Bloom filter. Section IV describes the cached counting Bloom filter architecture (CCBF). Section V presents our simulation results. Section VI, we draw the overall conclusions.

II. RELATED WORK

In this section, we present efforts related to our work. In [5], a cache design based on the standard Bloom filter is investigated and has been extended to support aging (adding the ability to evict stale entries from the cache), bound misclassification rates, and use multiple binary predicates. It examines the exact relationship between the size and dimension of the number of flows that can be supported and the misclassification probability incurred. Additionally, it presents extensions for gracefully aging the cache overtime to minimize misclassification. In [6][10], an extended version of the Bloom filter is considered. It presents a novel hash table architecture and lookup algorithm and converts a Bloom filter into a counting Bloom filter and associated hash bucket which improves the performance over a standard hash table by reducing the number of memory accesses needed for the most time-consuming lookups. Our approach finds the mathematical model for cache architecture based on the value of the counters in the counting Bloom filters.

III. COUNTING BLOOM FILTER

In this section, we present the standard Bloom filter, the counting Bloom filter concepts, and afterward, describe the pruning procedure in counting Bloom filters.

A. Standard Bloom Filter

A Bloom filter is a simple space efficient randomized data structure for representing a set in order to support membership queries. Burton Bloom introduced Bloom filters in the 1970s [3]. A set S $(x_1, x_2, ..., x_n)$ of n elements 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 element $x \in S$, 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 yield 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[8][10]. The false positive probability is given as follows:

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

In this equation, n represents the number of elements, 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:

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

An example of the Bloom filter is depicted in Figure 1.

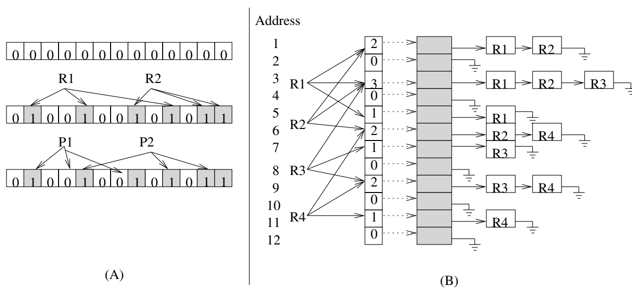

Fig. 1. (A) An example of a Bloom filter (B) The hash table architecture using counting Bloom filters for four items.

Figure 1(A) depicts the creation of a Bloom filter for a set of two items $R1$ and $R2$ and the subsequent testing whether $P1$ and $P2$ are part of the set. Each item R_i is hashed k times (using k independent hashing functions) and the corresponding bits are set to 1. To check whether $P1$ or $P2$ are member of the set, they are hashed with the same k hashing functions to determine the locations in the array to check whether these locations were set. For $P1$, it is clear that it is not part of the set.

B. Counting Bloom Filter

The previously discussed Bloom filter works fine when the members of the set does not change over time. When they do, adding items requires little effort since it only requires hashing the additional item and setting the corresponding bit locations in the array. On the other hand, removing an item conceptually requires unsetting the ones in the array, but this could inadvertently lead to removing a '1' that was the result of hashing another item that is still member of the set. To overcome this problem, the counting Bloom filter has been introduced[7]. In the counting Bloom filter, each bit in the array is replaced by a small counter. when inserting an item, each counter indexed by the corresponding hash value is incremented, therefore, a counter in this filter essentially give us the number of items hashed to it. When an item is deleted, the corresponding counters are decremented. Figure 1(B) depicts the results of hashing four items. Additionally, in this figure, we introduce the concept of *buckets* that are pointed to by the counters storing the items of the set.

$C(i)$ is used to denote the count associated with the ith counter. Considering a Bloom filter for n items, with k hashing functions, and m counters, the probability that the ith counter is incremented j times is given as a binomial random variable in the following:

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

An n-bit counter will overflow if and only if it reaches a value of 2^n. The analysis performed by Fan, et.al. [7] shows that a 4-bit counter is adequate for most applications. The probability for 4-bit counter to overflow is:

$$p\left(max_i \langle c(i) \geq 16 \rangle\right) \leq 1.37 \times 10^{-15}.m \quad (4)$$

Based on a counting Bloom filter, we compute k hashing functions $h_1(), ..., h_k()$ over an input item and increment the related k counters indexed by these hash values. Subsequently, we store the item in the lists associated with each of the k buckets hence a single item is stored k times in memory. In the mentioned approach, we need to maintain up to k copies of each item requiring k times more memory compared to a standard hash table. However, in a Bloom filter only one copy is accessed while the other $(k-1)$ copies of item are never accessed, therefore, the memory requirement can be minimized in the mentioned architecture, resulting in the pruned counting Bloom filter. The pruned counting Bloom filter for Figure 1(B) is depicted in Figure 2.

There are two alternatives to perform pruning on a Bloom filter. The first method entails the creation of a

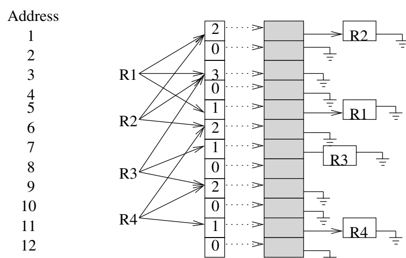

Fig. 2. The hash table architecture using pruned counting Bloom filters.

regular Bloom filter. Afterwards, the regular Bloom filter is pruned by only maintaining those items in the buckets that are pointed to by the counter with the lowest value, i.e., all other already inserted items in the other buckets are deleted. In the event two counters have to the lowest value, the bucket pointed to by the counter with the lowest index is chosen. This method is simple, but it requires storage of n copies of each item in the set in the creation of the pruned Bloom filter and additionally a special procedure to delete redundantly stored items. In second method [2][10], the memory requirement is equal to n, which is the number of items, and thereby reducing the memory requirements (compared to the first method). In the meantime, it involves more computations to create the pruned Bloom filter in compared to the previous method. It must be noted that during the pruned counting Bloom filter creation, the counter values are not changed. Consequently, after pruning, the counter does not express the number of items in the list and is larger than or equal to the number of items in each bucket.

IV. Cached Counting Bloom Filter Concept and Analysis

In this section, we present the concept of the cached counting Bloom filter and its analysis.

A. Cached Counting Bloom Filter Concept

According to definition of a Bloom filter, the number of hashing functions can be expressed as:

$$k = g\frac{m}{n} \quad (5)$$

where the value of g changes for different Bloom filter configurations. The optimal value for g to have a minimum false positive rate is $g = ln(2)$. Using Eq. 1 and Eq. 5, the false positive rate for different value of g is depicted in Figure 3.

After the substitution of Eq. 5 in Eq. 3, we obtain the following equation:

$$p\left(c(i) = j\right) = \begin{pmatrix} gm \\ j \end{pmatrix} \left(\frac{1}{m}\right)^{j} \left(1 - \frac{1}{m}\right)^{mg-j} \quad (6)$$

Using Eq. 6, we can compute the probability of incrementing of the ith counter for different values of g and m. Using Eq. 6, the counter probability distribution for different counting Bloom filter configurations is depicted in Figure 4.

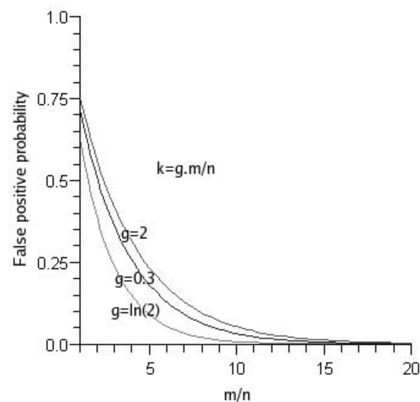

Fig. 3. False positive probability for different configurations.

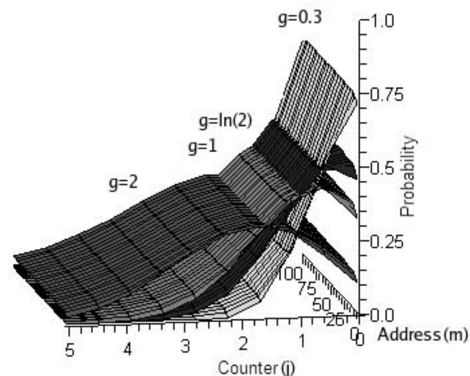

Fig. 4. The counter probability distribution for different configurations in counting Bloom filters.

Based on the figure, when $g \leq ln(2)$ the value of the counters with non-zero probability changes between 0 and 3, and when $g > ln(2)$ the value of the counters with non-zero probability is increased (for $g = 2$, the value of the counters changes between 0 and 5). Therefore, we can utilize a multi-level caching memory to store the items. We introduce the cached counting Bloom filter a Bloom filter with each counter pointing to the level corresponding to its counter value and with level l containing l buckets.

B. Counting Cached Bloom Filter Analysis

In this section, we present the analysis of the cached counting Bloom filter. The number of accesses to the memory depends on the fact whether the Bloom generates a 'positive' or 'negative' result. For the negative case, no accesses to the memory is needed since it certain that they are not in the original set. For the positive case, still it must be verified whether the item in question is a member or not (false positive). Consequently, we assume in the analysis that all tests are on different elements which would result in the testing of n elements (the same number of items in the original set). The number of accesses in a standard Bloom filter is $nk(1 + p_f)$ memory accesses, where n represents the number of items, k represents the number of hashing functions and p_f is false positive probability. The l-level

cached counting Bloom filter architecture is depicted in Figure 5.

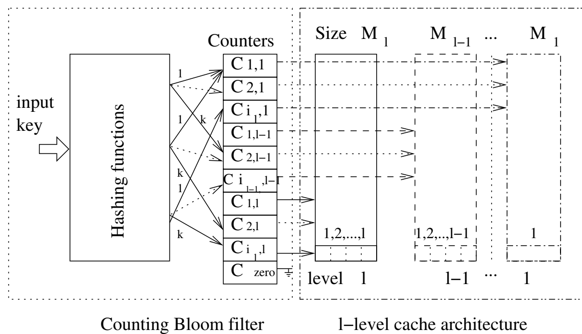

Fig. 5. The l-level cached counting Bloom filter architecture.

In this figure, $C(i_l, l)$ represents the counter with the value "l" pointing to location i_l within cache level "l". Therefore, the values of $C_{1,1}, ..., C_{i_1,1}$ are equal to 1, the values of $C_{1,l-1}, ..., C_{i_{l-1},l-1}$ are equal to $l-1$ and the values of $C_{1,l}, ..., C_{i,l}$ are equal to l. C_{zero} shows the counters with value 0 and do not point to any bucket in the cache memory. These counters are represented by C_{zero}. M_l represents the size of the cache memory in level l. From the Figure 5, the number of accesses in l-level CCBF is equal to summation of accesses in each level as follows:

$$\begin{gathered} Number\ of\ accesses\ in\ l-level\ CCBF = \\ (N_1 + ... + N_i + ... + N_l) \end{gathered} \quad (7)$$

In this equation, N_i represents the number of accesses in level i. Using the Eqs. 3 and 7, the number of accesses in CCBF is extended as follows:

$$\begin{gathered} Number\ of\ accesses\ in\ l-level\ CCBF = \\ A\left(p(j=1) + \frac{p(j=2)}{2} + ... + \frac{p(j=l)}{l}\right) = \\ A\begin{pmatrix} A \\ 1 \end{pmatrix}\left(\frac{1}{m}\right)\left(1-\frac{1}{m}\right)^{A-1} + ...+ \\ A\begin{pmatrix} A \\ l \end{pmatrix}\left(\frac{1}{l}\right)\left(\frac{1}{m}\right)^{l}\left(1-\frac{1}{m}\right)^{A-l} \quad (with\ A = nk(1+p_f)) \end{gathered} \quad (8)$$

For the large values of m, the Eq. 9 can be utilized:

$$\left(1-\frac{1}{m}\right)^{nk} \cong e^{\frac{-nk}{m}} \quad (9)$$

Using Eq. 9, we can rewrite the Eq. 8 as follows:

$$\begin{gathered} Number\ of\ accesses\ in\ l-level\ CCBF = \\ A\left(p(j=1) + \frac{p(j=2)}{2} + ... + \frac{p(j=l)}{l}\right) = Ae^{\frac{-A}{m}} \\ \left(\left(\frac{1}{m}\right)\begin{pmatrix} A \\ 1 \end{pmatrix} + ... + \begin{pmatrix} A \\ l \end{pmatrix}\left(\frac{1}{l}\right)\left(\frac{1}{m}\right)^{l}\right) \cong \\ Ae^{\frac{-A}{m}}\left(\sum_{i=1}^{l}\frac{1}{ii!}\left(\frac{A}{m}\right)^{i}\right) \quad (with\ A = nk(1+p_f)) \end{gathered} \quad (10)$$

If we assume that $\frac{m}{n} = c$ then we can rewrite Eq. 10 as follows:

$$\begin{gathered} Number\ of\ accesses\ in\ l-level\ CCBF = \\ nk(1+p_f)e^{\frac{-k}{c}(1+p_f)}\left(\sum_{i=1}^{l}\frac{1}{ii!}\left(\frac{k(1+p_f)}{c}\right)^{i}\right) \end{gathered} \quad (11)$$

After the normalization to $nk(1+p_f)$ the number of accesses is expressed as function of c and k as follows:

$$\begin{gathered} Number\ of\ accesses\ in\ l-level\ CCBF = \\ e^{\frac{-k(1+p_f)}{c}}\left(\sum_{i=1}^{l}\frac{1}{ii!}\left(\frac{k(1+p_f)}{c}\right)^{i}\right) \end{gathered} \quad (12)$$

Designing an l-level CCBF is impractical. Therefore, we propose to limit the number of levels to 3 based on our observation from Eq. 6 and Figure 4 that the counter values are not likely to be larger than 3. More precisely, levels 1 and 2 (containing 1 and 2 buckets, respectively) store the elements for the counters with values 1 and 2, respectively. Level 3 stores the elements for counters with value 3 or larger. As the counters with larger than 3 values require more storage, the elements are stored over multiple rows in the third level of the CCBF (*segmentation*). The 3-level cache architecture is depicted in Figure 6.

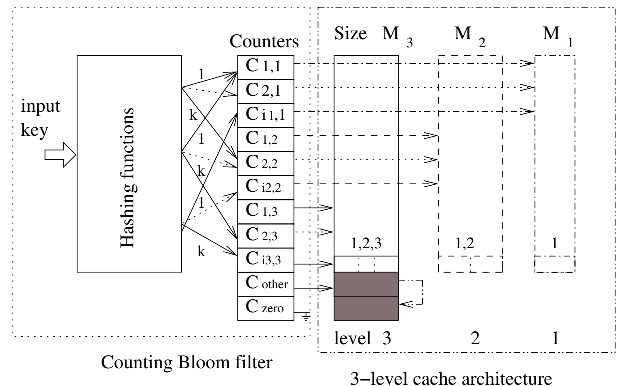

Fig. 6. The 3-level cached counting Bloom filter architecture.

In Figure 6, the values of $C_{1,1}, ..., C_{i_1,1}$ are equal to 1, the values of $C_{1,2}, ..., C_{i_2,2}$ are equal to 2 and the values of $C_{1,3}, ..., C_{i_3,3}$ are equal to 3. C_{other} represents the counters with values larger than three and, therefore, they point to a storage within level 3 of the CCBF. Figure 6 highlights the mentioned segmentation. C_{zero} represents the counters with their value being zero. In the following, we analyze the effects of the items with counter values larger than three. The number of accesses in 3-level CCBF is equal to number of accesses in the levels 1, 2, and 3 combined. The number of accesses in third level of cache can be computed as a summation of the number of counters with value 3 and larger. Therefore, the number of accesses in a 3-level CCBF is as follows:

$$\begin{gathered} Number\ of\ accesses\ in\ 3-level\ CCBF = \\ A\left(p(j=1) + \frac{p(j=2)}{2} + \frac{p(j\geq 3)}{3}\right) = \\ A\left(p(j=1) + \frac{p(j=2)}{2} + \frac{p(j=3)}{3}\right) + \\ \left\lceil\frac{1}{3}p(j=4)A\right\rceil + ... + \left\lceil\frac{1}{3}p(j=l)A\right\rceil = \\ A\begin{pmatrix} A \\ 1 \end{pmatrix}\left(\frac{1}{m}\right)\left(1-\frac{1}{m}\right)^{A-1} + \\ A\begin{pmatrix} A \\ 2 \end{pmatrix}\left(\frac{1}{2}\right)\left(\frac{1}{m}\right)^{2}\left(1-\frac{1}{m}\right)^{A-2} + \\ A\begin{pmatrix} A \\ 3 \end{pmatrix}\left(\frac{1}{3}\right)\left(\frac{1}{m}\right)^{3}\left(1-\frac{1}{m}\right)^{A-3} + \\ \sum_{i=4}^{l}\left\lceil\begin{pmatrix} A \\ i \end{pmatrix}\left(\frac{1}{3}\right)\left(\frac{1}{m}\right)^{i}\left(1-\frac{1}{m}\right)^{A-i}A\right\rceil \\ (with\ A = nk(1+p_f)) \end{gathered} \quad (13)$$

Using Eq. 9, the Eq. 13 is rewritten as follows:

$$\begin{gathered} Number\ of\ accesses\ in\ 3-level\ CCBF \cong \\ Ae^{\frac{-A}{m}}\left(\frac{A}{m}+\frac{1}{2*2!}(\frac{A}{m})^2+\frac{1}{3*3!}(\frac{A}{m})^3\right)+ \\ \sum_{i=4}^{l}\left\lceil\frac{1}{3i!}e^{\frac{-A}{m}}\left(\frac{A}{m}\right)^i A\right\rceil \quad (with\ A=nk(1+p_f)) \end{gathered} \tag{14}$$

According to definition and properties of the *ceiling function*, this relation $x \leq \lceil x \rceil < x+1$ is true for each x in real numbers, therefore, we can write for the second term in Eq. 14 to the following inequality:

$$\begin{gathered} \sum_{i=4}^{l}\frac{1}{3i!}e^{\frac{-A}{m}}\left(\frac{A}{m}\right)^i A \leq \sum_{i=4}^{l}\left\lceil\frac{1}{3i!}e^{\frac{-A}{m}}\left(\frac{A}{m}\right)^i A)\right\rceil < \\ \sum_{i=4}^{l}\frac{1}{3i!}e^{\frac{-A}{m}}\left(\frac{A}{m}\right)^i A+l-4 \quad (with\ A=nk(1+p_f)) \end{gathered} \tag{15}$$

Using this inequality, we can rewrite Eq. 15 as follows:

$$\begin{gathered} Ae^{\frac{-A}{m}}\left(\frac{A}{m}+\frac{1}{2*2!}(\frac{A}{m})^2+\frac{1}{3*3!}(\frac{A}{m})^3\right)+\sum_{i=4}^{l}\frac{1}{3i!}e^{\frac{-A}{m}}\left(\frac{A}{m}\right)^i A \\ \leq Number\ of\ accesses\ in\ 3-level\ CCBF \\ < Ae^{\frac{-A}{m}}\left(\frac{A}{m}+\frac{1}{2*2!}(\frac{A}{m})^2+\frac{1}{3*3!}(\frac{A}{m})^3\right)+ \\ \sum_{i=4}^{l}\frac{1}{3i!}e^{\frac{-A}{m}}\left(\frac{A}{m}\right)^i A+l-4 \quad (with\ A=nk(1+p_f)) \end{gathered} \tag{16}$$

After the normalization to $A = nk(1+p_f)$ (number of accesses in standard Bloom filter) Eq. 15 is rewritten as follows:

$$\begin{gathered} e^{\frac{-A}{m}}\left(\frac{A}{m}+\frac{1}{2*2!}(\frac{A}{m})^2+\frac{1}{3*3!}(\frac{A}{m})^3\right)+\sum_{i=4}^{l}\frac{1}{3i!}e^{\frac{-A}{m}}\left(\frac{A}{m}\right)^i \\ \leq Number\ of\ accesses\ in\ 3-level\ CCBF \\ < e^{\frac{-A}{m}}\left(\frac{A}{m}+\frac{1}{2*2!}(\frac{A}{m})^2+\frac{1}{3*3!}(\frac{A}{m})^3\right)+ \\ \sum_{i=4}^{l}\frac{1}{3i!}e^{\frac{-A}{m}}\left(\frac{A}{m}\right)^i+\frac{l-4}{A} \quad (with\ A=nk(1+p_f)) \end{gathered} \tag{17}$$

In practice, we can observe that the value of $l \ll nk(1+p_f)$, since $\frac{l}{nk(1+p_f)} \approx 0$, therefore we can write the number of accesses in 3-level CCBF as follows:

$$\begin{gathered} Number\ of\ accesses\ in\ 3-level\ CCBF \cong \\ e^{\frac{-nk(1+p_f)}{m}}\left(\frac{nk(1+p_f)}{m}+\frac{1}{2*2!}(\frac{nk(1+p_f)}{m})^2+\frac{1}{3*3!}(\frac{nk(1+p_f)}{m})^3\right)+ \\ \sum_{i=4}^{l}\frac{1}{3i!}e^{\frac{-nk(1+p_f)}{m}}\left(\frac{nk(1+p_f)}{m}\right)^i \end{gathered} \tag{18}$$

After substitution of $\frac{m}{n}$ with c the number of accesses in the 3-level CCBF is written as follows:

$$\begin{gathered} Number\ of\ accesses\ in\ 3-level\ CCBF \cong \\ e^{\frac{-k(1+p_f)}{c}}\left(\frac{k(1+p_f)}{c}+\frac{1}{2*2!}(\frac{k(1+p_f)}{c})^2+\frac{1}{3*3!}(\frac{k(1+p_f)}{c})^3\right)+ \\ \sum_{i=4}^{l}\frac{1}{3i!}e^{\frac{-k(1+p_f)}{c}}\left(\frac{k(1+p_f)}{c}\right)^i \end{gathered} \tag{19}$$

In the pruned counting Bloom filter for each item k hashing functions are computed and only one item is stored in the memory, therefore, the CCBF can be applied for pruning procedure. The memory organization in the pruned CCBF can be seen as a memory that is organized by a counting Bloom filters with one hashing function. In the other words, the cache architecture for counting Bloom filter with one hashing function can assumed as a good approximation for cached pruned counting Bloom filter. In this way, we can optimize the number of accesses using cache architecture and memory usage by pruning procedure.

In the following, we evaluate the size of the different cache levels in the CCBF architecture. In short, the size of each cache level is equal to the multiplication of nk and the probability of each counter value in the CCBF. The size of each cache level in l-level CCBF is expressed as follows:

$$\begin{gathered} The\ size\ of\ level\ \text{j}\ within\ l-level\ CCBF = \\ nk.p\left(c(i)=(level\ number)\right)= \\ nk.p\left(c(i)=j\right)=nk\begin{pmatrix} nk \\ j \end{pmatrix}\left(\frac{1}{m}\right)^j\left(1-\frac{1}{m}\right)^{nk-j} \end{gathered} \tag{20}$$

In this equation, j is level number. Using Eqs. 7 and 9, we can rewrite the previous equation as follows:

$$\begin{gathered} The\ size\ of\ level\ j\ in\ l-level\ CCBF \cong \\ nke^{\frac{-k}{c}}\left(\frac{1}{j!}\left(\frac{k}{c}\right)^j\right)\ where\ j\ is\ level\ number \end{gathered} \tag{21}$$

Using Eq. 21, the total size of l-level CCBF cache after normalization to nk (size of a standard Bloom filter) is:

$$\begin{gathered} The\ total\ size\ of\ l-level\ CCBF = nk\sum_{j=1}^{l}(p(c(i)=j) \cong \\ e^{\frac{-k}{c}}\left(\sum_{j=1}^{l}\frac{1}{j!}\left(\frac{k}{c}\right)^j\right) \end{gathered} \tag{22}$$

Applying this equation to the 3-level CCBF case, results in the following sizes of the 3 levels (keeping in mind 4-bit counter, the with l being 16):

$$\begin{gathered} The\ level\ 1 = e^{\frac{-k}{c}}\frac{k}{c} \quad The\ level\ 2=\frac{1}{2}e^{\frac{-k}{c}}\left(\frac{k}{c}\right)^2 \\ The\ level\ 3=e^{\frac{-k}{c}}\left(\sum_{j=3}^{l}\frac{1}{j!}\left(\frac{k}{c}\right)^j\right) \end{gathered} \tag{23}$$

V. Simulation Results

In this section, we present the simulation results of CCBF architecture. The simulation results were generated using Maple v.10.0. The number of accesses for different CCBF configurations is depicted in Figure 7.

The l-level CCBF result based on Eq. 11 is depicted in Figure 7 (A). In this figure, the x axis is labeled by m/n that m shows address space size and n shows number of items. y axis is labeled by k number of hashing functions and z axis is labeled by number of accesses that is normalized to nk. We can observe that, the l-level CCBF decreases the number of accesses at least by 51.6% in compared to standard Bloom filter. The 3-level CCBF results based on Eq. 19 are depicted in Figure 7 (B). We can observe that the 3-level CCBF decreases the number of accesses at least by 51.3% in compared to standard Bloom filter. It is clear that, the results can explain the pruning of counting Bloom filter. The comparison between 3-level and l-level CCBF is depicted in Figure 7 (C). We can observe that, the difference of two approaches is negligible. The size of different levels in 3-level CCBF is depicted in Figure 8 (A).

The increasing of $\frac{m}{n}$ will decrease size of different cache levels, since the more addresses are stored in level 0 (level 0 represents addresses from address space that the counter values are 0). Based on the figure, when $\frac{m}{n}$ is small the size of level 3 is increased and the size of levels 1 and 2 is decreased. From the Figure 8 (A), we can find the area to have a balance between different cache level in 3-level CCBF

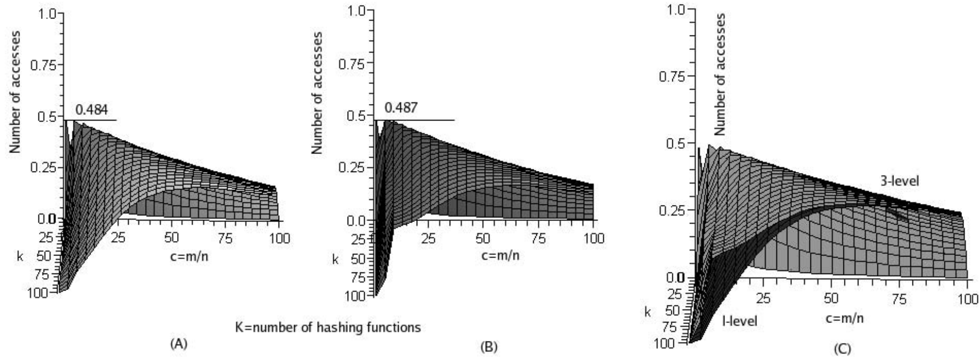

Fig. 7. The number of accesses in CCBF normalized to number of accesses in standard Bloom filter $(nk(1+p_f))$. (A) The number of accesses in a l-level CCBF. (B) The number of accesses in a 3-level CCBF. (C) The comparison of a l-level and a 3-level CCBF.

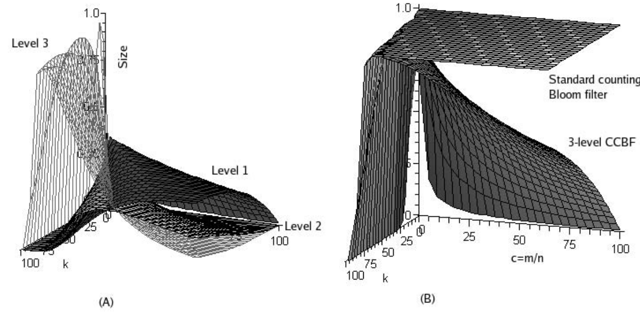

Fig. 8. The cache size in a CCBF normalized to size of standard Bloom filter $(nk(1+p_f))$. (A) Different levels size in a 3-level CCBF. (B) The total cache size in a 3-level CCBF.

by finding intersecting area in different cache levels. To find the intersecting area, we solve the following equations.

$$nke^{\frac{-k}{c}}\left(\sum_{j=3}^{l}\frac{1}{j!}\left(\frac{k}{c}\right)^j\right) = nke^{\frac{-k}{c}}\frac{k}{c}$$
$$nke^{\frac{-k}{c}}\left(\sum_{j=3}^{l}\frac{1}{j!}\left(\frac{k}{c}\right)^j\right) = nke^{\frac{-k}{c}}\left(\frac{k}{c}\right)^2 \qquad (24)$$

The final results of Eq. 24 are: $k = 1.87\frac{m}{n}$ and $k = 1.79\frac{m}{n}$. With substitution these values in Eq. 19, we can observe that the number of accesses is decreased by 52% using 3-level CCBF. The total size for 3-level CCBF in compared with standard Bloom filter is depicted in Figure 8(B).

VI. OVERALL CONCLUSIONS

In this paper, we presented a new approach to embed a cache in a counting Bloom filter (CCBF). Using the counting Bloom filter property, the number of accesses and sizes of cache levels in the CCBF architecture were investigated. We concluded from the results that incorporating a cache to Bloom filters will improve the performance of Bloom filter by at least 51.3% (in terms of memory accesses) compared to a standard Bloom filter. We expect this approach to be useful in the design of high performance memory architectures utilized in network processors and related applications such as packet classification and web caching.

REFERENCES

[1] 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*, pages 223–232, November 2006.

[2] 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)*, pages 70–76, February 2007.

[3] B. H. Bloom. "Space /Time Trade-offs in Hash Coding with Allowable Errors". *Communication of the ACM*, 13(7):422–426, July 1970.

[4] A. Broder and M. Mitzenmacher. "Network Applications of Bloom Filters: A Survey". In *Proc. 14'th Annual Allerton Conf. on Communication, Control, and Computing*, pages 636–646, October 2002.

[5] F. Chang, F. Wu-chang, and L. Kang. "Approximate Caches for Packet Classification". In *23'th Annual Conf. of the IEEE, INFOCOM*, pages 2196–2207, March 2004.

[6] S. Dharmapurikar, H. Song, J. Turner, and J. Lockwood. "Fast Packet Classification Using Bloom Filters". Technical Report 27, Department of Computer Science And Engineering, Washington University in St. Louis, May 2006.

[7] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. "Summary Cache: A Scalable Wide-Area (WEB) Cache Sharing Protocol". *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[8] 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)*, pages 91–103, October 2005.

[9] J. Mudigonda, H. M. Vin, 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)*, pages 1–10. ACM Press, October 2005.

[10] H. Song, J. Turner, S. Dharmapurikar, 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*, pages 181–192, August 2005.