Blooming Trees: Space-Efficient Structures for Data Representation

Domenico Ficara, Stefano Giordano, Gregorio Procissi, Fabio Vitucci
Dept. of Information Engineering, University of Pisa, ITALY
Email: {domenico.ficara, stefano.giordano, gregorio.procissi, fabio.vitucci}@iet.unipi.it

Abstract—A Bloom Filter is an efficient randomized data structure for membership queries on a set with a certain known false positive probability. A Counting Bloom Filter (CBF) allows the same operations on dynamical sets that can be updated via insertions and deletions with larger memory requirements.

This paper presents a novel hierarchical data structure, called Blooming Tree, that replicates the functionalities of a CBF with lower memory consumption and tunable false positive probability. The hierarchical multi-layer design of Blooming Trees allows for distributing the structure in different memory levels, thus exploiting small but fast on-chip memories for most frequently accessed substrucures. The proposed algorithm is compared to previous existing schemes on a target platform: Intel IXP2XXX Network Processors (NPs).

Index Terms—Counting Bloom Filter, Network Processors, Binary Trees, MultiLayer Structure

I. INTRODUCTION

Today streamed data processing is an issue in many areas related to computer applications. In particular, detecting whether an item belongs to a set is a challenging task, especially when the amount of data is very large and rapidly changes.

A Bloom Filter (BF) is a simple data structure for information representation. Since it is a randomized method based on hash functions, it allows for false positives; however, the advantage of the space savings often outweighs such a drawback. BFs were introduced by Burton Bloom [1] in the 1970s for database applications, but they have recently received a significant attention in the networking area [2].

BFs are not usable for sets that change over time. For instance, deletion cannot be done by simply changing ones back to zeros, as a single bit may correspond to multiple elements. Therefore, Counting Bloom Filters (CBFs) have been designed [3]. They use fixed size counters instead of single bits of presence. When an item is inserted (deleted), the corresponding counter is incremented (decremented).

This paper proposes a novel data structure with the same functionalities of a CBF that allows a proper tuning the probability of false positives as well as the probability of overflow. The structure is allotted in different layers, thus exploiting the built-in memory hierarchy of many packet processing systems. Because of the similarities with binary trees and the tunable multilayer design, we call it Blooming Tree (BT). A “naive” construction of BT reduces the memory consumption at least of a factor of $2/\ln 2 \simeq 2.88$ times compared to that of standard CBF, while an optimized version achieves a saving of up to $4/\ln 2 \simeq 5.75$ times.

The following section presents the most important works on CBFs. Section III describes the Blooming Tree structure and section IV an optimized version, which has the aim of a further memory saving. Finally, a simulative comparison among our algorithms and the previous ones is presented in section V, by adopting Intel IXP2800 as referential hardware platform.

II. PREVIOUS WORKS

A Bloom Filter represents a set $S$ of $n$ elements from a universe $U$ by using an array of $m$ bits, denoted as $B[1],...,B[m]$, initially all set to 0. The filter uses $k$ independent hash functions $h_1,...,h_k$ that independently map each element in the universe to a random number uniformly over the range. For each element $x$ in $S$, the bits $B[h_i(x)]$ are set to 1 for $1 \leq i \leq k$ (a bit can be set to 1 multiple times).

To answer a query of the form “Is $y$ in $S$?”, one checks whether all $B[h_i(y)]$ are set to 1. If not, $y$ is not a member of $S$, by construction. If all $B[h_i(y)]$ are set to 1, it is assumed that $y$ is in $S$; hence, a BF may yield a false positive. The probability of a false positive ($f$) can be opportunely tuned by choosing the proper values for $m$ and $k$. The optimal value for $k$, in terms of $f$, is $(m/n)\ln 2$, that yields $f = 2^{-k}$ [3].

As said in the previous section, BFs do not allow insertion and deletion of items in the set. Therefore, CBFs have been introduced, which use bins instead of single bits of presence. They present the problem of counter overflow, which results in a lack of accuracy of stored information. Moreover, the use of a fixed number of bits per bin causes a memory wastage.

Many improvements to CBFs have been proposed. In [4], the authors show that unbalancing the number of ones and zeros in a standard BF can help achieving a good compression ratio before transmission (e.g. for web-caching application).

Spectral Bloom Filters (SBFs) [5] are an extension of standard BFs to multi-sets, allowing estimates of the multiplicities of individual items. SBFs dynamically vary the size of their counters in order to use a smaller number of bits, but they include additional slack bits and complex index structures.

Dynamic Count Filters (DCFs) [6] consist of two vectors: the first one is a basic CBF with counters of fixed size, the second one is the Overflow Vector (OV), which has a counter for each element of the first vector that keeps track of the number of times that this element suffers from an overflow. To avoid saturation, the size of counters in OV changes dynamically, though this implies that each single update requires to rebuild the whole structure. Moreover, the
decision of maintaining the same size for all the counters in OV (to allow direct access) entails that many bits are not used.

D-left CBFs (d-CBFs) [7] are simple alternatives to CBFs based on d-left hashing and fingerprints. To the best of our knowledge, they represent the best solution in literature in terms of memory consumption. However, d-CBFs have the limitation of potential counter overflow and the need of an additional fingerprint for each bin in the data structure.

Finally, ML-CCBF, a multilayered structure that exploits memory hierarchy of modern systems, is presented in [8]. It is based on Huffman coding for a further bins compression.

The memory utilization is the parameter that is better focused on this work. There are several situations where network bandwidth is still expensive and transmission size becomes a fundamental parameter. Moreover, although memory appears plentiful today, there are many hardware architectures used in network devices (e.g. NPs) that could benefit from the use of very space-efficient data structures. Indeed, memory saving can greatly speedup a device if it dramatically reduces the number of accesses to slower off-chip memories. All these issues have led this research activity: the target is to propose an efficient data structure which outperforms the previous solutions, especially in terms of memory consumption.

III. BLOOMING TREE

The main idea behind Blooming Trees is the construction of a binary tree upon a plain BF, thus creating a multilayered structure where each layer represents a different depth-level of tree nodes. The aim is to achieve both low false positive probability $f$ and low memory requirements. The drawback is the increased cost in lookup operation, that can be mitigated by the low memory consumption that enables the deployment of the proposed structure in faster on-chip memories.

To begin the description, we first show the construction of BT with no optimization (we shall call it NBT, i.e. Naive BT); an optimized version of BTs will be elaborated upon in sec.IV.

To build a NBT for $n$ elements, $L + 2$ layers are defined:

- a plain BF ($B_0$) with $k_0$ hash functions $h_j$ ($j = 1...k_0$) and $m$ bins such that $m = nk_0 / \ln 2$;
- $L$ layers ($B_1...B_L$), each composed by $m_i$ ($i = 1...L$) blocks of $2^b$ bits ($m_i$ are not fixed, but depend on the input set, in particular they are limited in case of many hash collisions);
- a final layer ($B_{L+1}$) composed by $c$-bits counters.

The $j$-th hash function $h_j$ provides a $log_2 m + L \times b$ bit long output: the first group ($s_{0,j}$) of $log_2 m$ bits is used to address the BF at layer 0, the other $L \times b$ bits are divided into $L$ substrings ($s_{1,j}...s_{L,j}$) of $b$ bits, one for each layer.

Let $\text{popcount}(B_i[i])$ be the number of ones in the bitmap $B_0[0]...B_i[i-1]$ and let us consider the simplest case $b = 1$ (this way, blocks become couples and substrings $s_{1,j}$ collapse into single bits). The lookup for an element $\sigma$ consists of a check on $k_0$ elements in the BF and an exploration of the corresponding $k_0$ “branches” of the Blooming Tree. As shown in pseudocode 1, we jump from layer $i$ to layer $i + 1$ by:

1. computing a popcount on layer $i$, that gives us the index of the couple to be observed in the layer $i + 1$;
2. checking the bit expressed by $s_{i,j}$; if $s_{i,j}$ is equal to 0, we check the first bit of the couple, otherwise the second;
3. processing the bit of the couple; if it is 0, then $\sigma$ is not in the set and the lookup result is NOT FOUND, otherwise the overall process must be iterated for the next layers.

Therefore, a lookup of an element requires, for each hash function, a popcount and a bit check on the hash and on the block; these operations are needed for each layer, until the result is found, thus resulting in a maximum number of operations equal to $k_0[\text{hash} + L(\text{popcount} + 2 \times \text{check})]$. However, the computational cost of a lookup is negligible as popcount and hash operations are supported by hardware in most modern NPs (such as the Intel IXP2800 [9]) and often replicated in multi-core architectures.

Algorithm 1 Pseudo-code for the lookup of element $\sigma$ in BT

```plaintext
for $i \leftarrow 1, k_0$ do
  $s_{0,i} \leftarrow s_{0,j}$
for $i \leftarrow 0, L + 1$ do
  if $B_i(x_i) = 0$ then
    return NOT FOUND
end if
if $x_i + 1 \leftarrow 2^b \times \text{popcount}(B_i[x_i]) + s_{i,j}$ then
  end for
else
  end for
```

An example (with $k_0 = 1$) of the lookup process is shown in fig. 1, where the tree structure of NBT is clear. For instance, let us observe the last bit of the BF, where two items collide. The popcount (equal to 3) leads to the proper block of layer $B_1$. The bit $s_1$ of the hash is equal to 0 for both the items, so the first bit of couple is set. Then, the items present a different $s_2$ bit of the hash and they split: in the fourth couple of layer $B_2$ (as indicated by the popcount on $B_1$) both the bits are set. Therefore two different bins in $B_3$ count the two items.

When inserting an element $\sigma$, we start from the standard BF ($B_0$). For each hash function $h_j(\sigma)$ ($j = 1...k_0$), we extract $x_0 = s_{0,j}$ and set $B_0[x_0]$. Then we compute $x_1 = \text{popcount}(B_0[x_0])$ and jump onto layer 1 at the $x_1$-th couple. Now we have two different possibilities according to the bit we just set in the BF. If it was already set (i.e., we have a collision in the BF), we set in the couple the position given

![Figure 1. An example of a Naive Blooming Tree with $b = 1$.](image-url)
by the next bit \((s_{1,j})\) of \(h_j(\sigma)\). If it was clear before setting, we allocate a couple in the \(x_{1}\)-th position (it is a right-shift by \(2^k\) positions) and set the position given by \(s_{1,j}\) in the new couple. Then, for all other layers, we repeat all the previous steps, as shown in pseudocode 2. Therefore the insertion of an element requires a maximum number of operations equal to \(k_0[\text{hash} + L(\text{popcount + shift + bitset})]\).

**Algorithm 2 Pseudo-code for the insertion of element \(\sigma\)**

1. for \(l = 1, k_0\) do
2. \(x_0 \leftarrow s_{0,l}\)
3. for \(i = 0, L\) do
4. \(\text{prev}_i \leftarrow B_i(x_i)\)
5. \(B_i(x_i) \leftarrow 1\)
6. \(x_{i+1} \leftarrow 2^b \times \text{popcount}(B_i(x_i))\)
7. if \(\text{prev}_i = 0\) then
8. \(B_{i+1}(x_i) + 2^b \times m_i \leftarrow B_{i+1}(x_i + 1)\)
9. \(m_i + 2^b \leftarrow m_i + 2^b\)
10. end if
11. \(x_{i+1} \leftarrow x_{i+1} + s_{1,i}\)
12. end for
13. \(B_{L+1}(x_{i+1}) \leftarrow B_{L+1}(x_{i+1}) + 1\)
14. end for

For the deletion of an element \(\sigma\) (see pseudocode 3), the corresponding counter in the layer \(B_{L+1}\) has to be found (by following the lookup process) and decremented. If the new counter value is equal to 0, its associated block of the previous layer has to be checked: if only the bit concerning \(\sigma\) is set, the overall block has to be removed (by a left-shift by \(2^b\) bits) and the same way the lower layer has to be processed. Otherwise, if other bits are set in the block, the deletion process ends.

**Algorithm 3 Pseudo-code for the deletion of element \(\sigma\)**

1. for \(l = 1, k_0\) do
2. \(x_0 \leftarrow s_{0,l}\)
3. for \(i = 0, L\) do
4. \(x_{i+1} \leftarrow 2^b \times \text{popcount}(B_i(x_i)) + s_{1,i}\)
5. end for
6. \(B_{L+1}(x_{i+1}) \leftarrow B_{L+1}(x_{i+1}) - 1\)
7. if \(B_{L+1}(x_{i+1}) = 0\) then
8. \(i \leftarrow L\)
9. while \(B_i(x_i \mod 2^b) = 1\) do
10. \(B_i(x_i) \leftarrow B_i(x_i + 2^b)\)
11. \(m_i \leftarrow m_i - 2^b\)
12. \(i \leftarrow i - 1\)
13. end if
14. \(B_0(x_0) \leftarrow 0\)
15. end while
16. end if
17. end for

A. Properties of Blooming Tree

By construction, in an NBT, layer \(i + 1\) has as many blocks as the number of ones in layer \(i\). Thus, the worst case for the overall structure size occurs whenever all the branches start at layer 0 (i.e.: there are no collisions in \(B_0\)). In this case \(2^b k_0\) bits are necessary for each layer. Thus, considering also layers \(B_0\) and \(B_{L+1}\), the size \(S_{\text{NBT}}\) can be bounded by:

\[
S_{\text{NBT}} \leq nk_0(1/\ln 2 + 2^b L + c)
\]

To compute the overflow probability \((P_m)\) at layer \(L+1\), we observe the collisions for each bit of all layers. It is equivalent to substituting each bit with a counter and to observing its value. Let us consider a given counter at layer \(i + 1\). It descends from a “parent” counter at layer \(i\). When jumping from layer \(i\) to layer \(i + 1\), some of the elements colliding at layer \(i\) may have a different hash substring, thus reducing the value of the counter at the next layer. Therefore the probability \(P_{i+1}(\sigma)\) of having a counter with value \(\varphi\) in layer \(i + 1\) is the sum of the probabilities that the parent counter on layer \(i\) has the value \(\varphi + j\) times the probability that the \((i + 1)\)-th hash substring is the same for \(\varphi\) elements and different for the other \(j\):

\[
P_{i+1}(\varphi) = \sum_{j=0}^{\infty} P_i(\varphi + j) \left(\frac{\varphi + j}{\varphi}\right) \left(\frac{1}{2^b}\right)^j \left(1 - \frac{1}{2^b}\right)
\]

The probability of collision occurrences in hash tables is generally approximated by a Poisson model [10]. Moreover, as stated in [7], such a model (with parameter \(\alpha_0 = \ln 2\) can be applied also to CBFs:

\[
P_i(\varphi) \approx \frac{e^{-\alpha_i} \alpha_i^\varphi}{\varphi!} = \text{Poisson}(\alpha_i, \varphi)
\]

By using (2) in (1), we can now compute \(P_{i+1}(\varphi)\):

\[
P_{i+1}(\varphi) \approx \sum_{j=0}^{\infty} e^{-\alpha_i} \alpha_i^{\varphi + j} \left(\frac{\varphi + j}{\varphi}\right) \left(\frac{1}{2^b}\right)^j \left(1 - \frac{1}{2^b}\right)
\]

\[
= e^{\alpha_i(1 - 2^b)} e^{-\alpha_i(\alpha_i / 2^b)^2} \approx e^{\alpha_i(1 - 2^b) 2^{-b^2}}
\]

The Poisson pmf is invariant with respect to a binomial transform such as (1) except for the parameter that is divided by \(2^b\); thus, \(P_{L+1}(\varphi)\) is a Poisson pmf as well.

Finally, by iterating (3) and by approximating

\[
\prod_{i=1}^{L} e^{\alpha_i(1 - 2^b)} \text{ with } e^{\alpha_0}, \text{ we obtain:}
\]

\[
P_{L+1}(\varphi) \approx e^{\alpha_0} P_0(\varphi)^{2L - \alpha_0} \approx \left(\ln 2 / 2^b\right)^\varphi \frac{\varphi!}{\varphi!}
\]

Eq. (4) states that, for reasonable values of \(L\) (e.g., \(L \geq 10\)), the probability \(P_{L+1}(\varphi + 1)\) becomes much smaller than \(P_{L+1}(\varphi)\). Therefore, the probability of overflow \(P_{L+1}(\varphi) \geq 2^{c - 1}\) can be safely approximated by:

\[
P_{L+1}(\varphi) \geq 2^{c - 1} \approx P_{L+1}(2^c - 1) \approx (\ln 2 / 2^b)^{2^c - 1} (2^c - 1)!
\]

Since in each layer a hash substring of \(b\) bits is used to choose the block at the next layer, the probability of false positives \(f\) for our BT decreases by a rate of \(2^b\) per layer. Then, considering also the false positives probability of the layer 0 (the BF), we have:

\[
f = 2^{-k_0 + Lb}
\]
To give an idea, a standard 4-bit CBF with the same \( f \) requires \( k_0 + L \) hash functions and \( S_{CBF} \) bits:

\[
S_{CBF} = 4n(k_0 + L)/\ln 2 \geq 4 \frac{1 + L/k_0}{1 + (2^bL + c)\ln 2} S_{NBT}
\]  

(7)

This means that a NBT can be more than \( 2/\ln 2 \approx 2.88 \) times smaller than its equivalent CBF.

IV. MEMORY OPTIMIZATION

An optimized version of BT is described in this section. It follows from three main observations about NBTs:

- as suggested by the Poisson approximation (2) and as shown in fig.1, \( P_f(1) \) gives the most relevant contribution for all layers; indeed, once there are no collisions in a certain block at layer \( u \), there are no collisions also in the corresponding blocks in all upper layers \( u + 1 \ldots L \), but we use \( 2^b(L - u) + c \) bits for those blocks;
- all blocks always have at least a bit set: a block with \( 2^b \) zeros (let us call it zero-block) has no meaning;
- looking up \( w \) layers yields \( f = 2^{-(k_0 + wb)} \).

Therefore, whenever there are no collisions in a block, a zero-block can be used to indicate this situation and stop the “branch” from growing. But we cannot stop the lookup there, since it would increase the probability of a false positive. The solution of the Optimized BT (OBT) is to add a bitmap and an array of hash substrings for each layer. The array of substrings for layer \( i \) is composed by all the \((L - i)\) hash substrings that complete the hash of the “branches” that stop at layer \( i \). In the bitmap (of \( m_i \) bits), the generic \( j \)-th bit is set if the \( j \)-th block has no collision (i.e. zero-block); this way it can be used to address the substring array (see fig. 2).

The optimization can be also done at runtime.

Obviously, operational routines change. As for lookup of an element \( \sigma \), whenever the \( x_j \)-th block is a zero-block, we compute \( y_i = \text{popcount}(\text{bitmap}[x_i]) \) and compare the last \((L - i)\) bits of the hash of \( \sigma \) with the \( y_i \)-th element in the \( i \)-th substring array. This way, the lookup becomes faster as zero-blocks are very likely to occur in any layer, thus avoiding all the steps required to jump up to layer \( L + 1 \).

The insertion routine, however, can be slightly slower since it must also be aware of zero-blocks. If we have no collisions at layer 0, we add a zero-block, we set the corresponding bit in the bitmap as well as the corresponding substring in the substring array. Instead, if there is a collision, we have to check the colliding elements and create the corresponding branches up to the layer (let us say \( j \)) where the hash substrings differ. At layer \( j + 1 \) we repeat the ordinary steps: add two zero-blocks, set the corresponding bits in the \( j \)-th bitmap and add the two hash substrings in the \( j \)-th substring array. The computational cost of deletion, in turn, is about the same of that of insertion since, again, zero-blocks require additional processing but reduce the amount of accesses to upper layer.

The average size of the overall structure is:

\[
S_{OBT} \simeq n k_0 (L + 2^b + (1 + 2^b-1)/\ln 2)
\]  

(8)

In fact, the first layer \( B_0 \) is constructed with \( m = n k_0 / \ln 2 \) bits and has, on average, \( m/2 \) ones. For each “1” present in \( B_0 \) we build a block \( (2^b \) bits) in \( B_1 \). Let us suppose that all the \( n k_0 \) branches start at layer \( B_2 \). This means that we have \( n k_0 \) zero-blocks in \( B_2 \), a bitmap of \( n k_0 \) bits and \( n k_0 \) entries (of \( L - 1 \) bits) in the hash substring array of layer \( 2 \). All these components sums up to \( n k_0 (L + 2^b + (1 + 2^b-1)/\ln 2) \) bits.

If we introduce a collision in \( B_2 \), the number of blocks and the bitmap length become \( nk_0 - 1 \) while the substring array is reduced by \( 2 \times (L - 1) \) bits. At layer \( B_3 \) we add 2 blocks, 2 bits in the bitmap and 2 elements of \( L - 2 \) bits in the substring array. Thus a collision adds \( 2^b - 1 \) bits. However, the higher the layer, the fewer the collisions and so the bits they introduce. Indeed, by means of simulations, eq. (8) proves to be very precise and it shows that, as \( L \) increases, an OBT can become up to \( 4/\ln 2 \approx 5.75 \) times smaller than an equivalent CBF.

Fig. 3 shows that the overall gain in size is evident for OBT as compared to NBT and dl-CBF. The figure reports the size for the above mentioned structures as a function of \(-\log_2(f)\) (i.e. the number of layers in OBT and NBT) for \( n = 2048 \).

High values of \( L \) increase the size but reduce the number of collisions and false positives. The size increment is however less than the one of a standard CBF in the same conditions. As for \( b \), the less memory expensive choice is \( b = 1 \), while \( b = 2 \) halves the amount of steps in the upper layers.
V. MEASUREMENTS

To evaluate the proposed algorithms and compare it to other known algorithms, the NP Intel® IXP2800 [9] has been taken as referential hardware architecture. NPs are platforms that offer high packet processing capabilities and combine the programmability of general-purpose processors with the performance typical of hardware-based solutions. The IXP2800 is a fully programmable NP, characterized by a hierarchy of processing units (a XScale core and 16 microengines MEv2) and memory devices (4KB of local memory and 16KB of scratchpad memory). The bigger the memory, the slower the access to it. The hierarchy of memory devices in the IXP2800 reflects the memory architecture of many systems, which present small fast memories and slower big ones. Therefore the results of our research are very general.

As shown in tab. I, the operations of the algorithms have been weighted in terms of clock cycles for microengines, according to the IXP2800 Reference Manual [9] and ignoring operations with negligible costs such as shift and popcount.

Each algorithm has been simulated and evaluated in terms of memory consumption and processing load. In simulation runs, the parameters are set to obtain about the same probability of false positives and overflow among the different algorithms. Moreover, each structure (and each substructure for hierarchical algorithms) was located in the fastest possible memory.

In the first simulation run, we set \( n = 8192 \). For NBT and OBT, \( L = 11 \), \( k_0 = 1 \), \( c = 3 \), and \( b = 1 \), thus obtaining \( P_{ov} = 10^{-28} \) and \( f = 0.24 \times 10^{-3} \). Regarding NBT, we have stored in local memory the first layers \( B_0 \) and \( B_1 \), which fill 2.9 KB. Then layers \( B_2 \ldots B_8 \) have been located in scratchpad memory and layers \( B_9 \ldots B_{11} \) in SRAM. For OBT, we put the first two layers (3.6 KB) in local memory and the other ones in scratchpad, including all bitmaps and substring arrays.

To obtain about the same probabilities, in a standard CBF (with 4 bits per bin) we set \( k = 12 \) (this way, \( P_{ov} = 1.5 \times 10^{-16} \) and \( f = 0.24 \times 10^{-3} \)). The overall structure has been located in SRAM, therefore lookup, insertion and deletion require \( k \) accesses to this memory and \( k \) hashing.

Finally, for dl-CBF we set \( k = 10 \), thus obtaining \( P_{ov} = 2.96 \times 10^{-23} \) and \( f = 1.46 \times 10^{-3} \). The data structure has been located in SRAM, thus processing an element requires 2\( k \) hashing and \( k \) accesses to SRAM.

From tab. II, it is clear that OBT is the best solution in terms of memory consumption. OBT outperforms also dl-CBF, which presents the best results in literature. Moreover, the multilayer structure of our solutions allows for a remarkable decrease of operational costs. For \( n = 8192 \), OBT shows, in comparison with dl-CBF, a reduction of 86% of clock cycles for lookup and 56% for insertion/deletion.

For \( n = 2048 \), the trends are the same: our solutions outperform the previous ones in terms of both memory consumption and operational load. The structure of OBT can be even completely located in local memory, thus drastically reducing operational costs (e.g., 98% less than dl-CBF for lookup).

VI. CONCLUSIONS

In this paper, a novel smart and efficient data structure has been presented. It offers the same functionalities of CBFs, i.e. the membership query on dynamical data sets. The underlying idea is to build a tree upon a standard Bloom Filter and to introduce a multilayered structure (to take advantage of the memory hierarchy of many systems, such as programmable routers and network processors). Moreover, an optimized version of our algorithm has been illustrated, with the purpose of further reduce the memory consumption.

A comparison among the proposed algorithms and the ones defined in literature has been performed by using Intel® IXP2800 as referential hardware platform. The outcomes show a clear memory saving (up to 4/ln 2 in comparison with standard CBF) and operational cost reduction of our solutions.

REFERENCES