This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

ABC: Adaptive Binary Cuttings for Multidimensional Packet Classification

Haoyu Song, *Member, IEEE*, and Jonathan S. Turner, *Fellow, IEEE, ACM*

***Abstract*—Decision tree-based packet classification algorithms are easy to implement and allow the tradeoff between storage and throughput. However, the memory consumption of these algorithms remains quite high when high throughput is required. The Adaptive Binary Cuttings (ABC) algorithm exploits another degree of freedom to make the decision tree adapt to the geometric distribution of the filters. The three variations of the adaptive cutting procedure produce a set of different-sized cuts at each decision step, with the goal to balance the distribution of filters and to reduce the filter duplication effect. The ABC algorithm uses stronger and more straightforward criteria for decision tree construction. Coupled with an efficient node encoding scheme, it enables a smaller, shorter, and well-balanced decision tree. The hardware-oriented implementation of each variation is proposed and evaluated extensively to demonstrate its scalability and sensitivity to different configurations. The results show that the ABC algorithm significantly outperforms the other decision tree-based algorithms. It can sustain more than 10-Gb/s throughput and is the only algorithm among the existing well-known packet classification algorithms that can compete with TCAMs in terms of the storage efficiency.**

***Index Terms*—Decision tree, packet classification.**

I. Introduction

PACKET classification plays an important role in both edge and core routers to provide advanced network services. Despite the vast body of existing work, packet classification remains an open and challenging problem. On the one hand, network security, network virtualization, and network quality of service (QoS) are the driving factors for large-scale packet classification involving thousands to tens of thousands of filters in a single router. On the other hand, increasing network traffic poses greater challenges for fast packet classification. Today, 10-Gb/s connections are common in edge and core networks while the 40- or even 100-Gb/s links are starting to be deployed at scale in operational networks. A moderate 10-GbE line card needs to classify 15 million packets per second.

Designing a general packet classification system requires significant engineering considerations. Although TCAMs have been widely adopted in many high-end solutions, there are scenarios where algorithmic solutions are preferred. If an algorithm can satisfy the worst-case throughput with a small amount of memory, it is not worth investing TCAMs due to the cost, power dissipation, and board footprint considerations. For algorithmic solutions, the data structure can be stored in different types of memory devices, such as DRAM, SRAM, or even embedded memory. Today's ASIC has been able to embed up to a few hundred megabits of memory on-chip (e.g., IBM's eDRAM). The flexibility offered by the algorithmic solutions has an important implication: smaller storage allows the use of faster (but more expensive) memory devices to boost the throughput. If everything can be squeezed on-chip, the system throughput is maximized accordingly. This is also in line with the trend for system-on-chip integration that can significantly boost the line-card port density.

However, memory consumption of the packet classification algorithms is often pessimistic partly because of the intrinsic complexity of the problem [1]. To compete with TCAMs, an algorithm needs to ensure that its storage is more scalable and actually cheaper. Although it is hard to measure with great accuracy, we can still take some hints from the cell density and the manufacturing cost of different technologies. For example, a TCAM cell typically uses 14–16 transistors to store a bit, an SRAM cell uses six transistors, and an SDRAM cell uses just one transistor and one capacitor. We can reasonably assume that a bit in a TCAM component costs about 10 bits in an SRAM component. Since a TCAM component usually consumes 18 B (144 bits) to store a 5-tuple filter, an SRAM-based algorithm should consume no more than 180 B per filter in order to compete with TCAM. Unfortunately, many well-known algorithms fail to pass this criterion. For example, the Recursive Flow Classification (RFC) algorithm is very fast, but it consumes more than 1600 B per filter for a filter set with only 600 filters [2]. Similar inefficiency exists in some other algorithms, such as the Cross-producting algorithm [3], the Bit Vector (BV) algorithm [4], and the Aggregated Bit Vector (ABV) algorithm [5]. They all suffer a significant storage penalty, even though their throughput is comparable to TCAMs.

The decision tree-based algorithms [6]–[8] offer more flexible control over the storage. A decision tree is built by splitting the filter set recursively based on partial filter information. The splitting stops when each subset contains fewer filters than a predefined bucket size. The filters in each of these subsets are organized in a priority list. Gupta and McKeown concluded that it is impossible to arrive at a practical worst-case solution for packet classification [6] due to the complexity of the underlying point location problem [1]. Hyafil and Rivest first proved that the problem of building optimal decision trees in the sense of minimizing the search steps is NP-complete [9]. Later, Murphy and McCraw proved that construction of storage-optimal decision trees is also NP-complete [10]. No matter what measurements

Manuscript received March 20, 2008; revised December 14, 2009 and December 08, 2011; accepted March 05, 2012; approved by IEEE/ACM Transactions on Networking Editor J. Yates.

H. Song is with Network Research, Huawei Technologies, Santa Clara, CA 95050 USA (e-mail: shykcl@gmail.com).

J. S. Turner is with the Computer Science and Engineering Department, Washington University in St. Louis, St. Louis, MO 63130 USA.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2012.2190519

1063-6692/$31.00 © 2012 IEEE

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

we take, the attempt to construct a globally optimal decision tree is intractable. For this reason, all previous algorithms are heuristic-based, which focus on the local optimality only. The packet classification is performed by testing the decision tree and linearly searching the stored filters when a leaf node is reached. The decision tree-based algorithms are easy to implement in both software and hardware, but their performance is very sensitive to the filter set structure. In this paper, we scrutinize the fundamental reasons of the memory and throughput inefficiency in decision tree constructions. Based on our observations, we change the decision-making criteria to better comply with the high-level design goal and introduce extra degrees of freedom that allow more intelligent space splitting. The new algorithm comes with three different variations that are able to scale to large filter sets and provide sufficient throughput for 10-GbE networks when using the commodity off-chip memory devices.

II. Related Work

The decision tree (DT) construction process is all about splitting the filter set recursively by using partial header information. It is crucial to limit the storage that is used to implement the data structure as well as the effort required to traverse the tree in order to find the best matching filter. At each tree node, a decision is made to split the current filter set S into a number of subsets $s_1, s_2, \ldots s_n$. Each subset represents a child node. We know that

$$s_1 \cup s_2 \cup \ldots \cup s_n = S.$$

If $|s_i| >$ bucket size for $\forall i$, we keep splitting s_i. Otherwise, the corresponding node becomes a leaf. It is common that $s_i \cap s_j \neq \emptyset$ and $|s_i| \neq |s_j|$ for some i and j, which makes the decision tree less efficient.

Woo proposed a generic approach to split the filter set [8]. At each tree node, a header bit that was not examined in any ancestor nodes is chosen. The current filters are then copied into one or both child nodes according to the value of the bit (i.e., "1," "0," or "don't care"). The bit selection criteria seek to minimize both the tree depth and the tree size. The process works as follows: At each tree node, all the current filters are evaluated first to get the statistics of the 1/0 distribution and the number of "don't care" specifications for every bit position. Then, the bit at the position with the fewest "don't care" specifications and the most uniform distribution of 1's and 0's is chosen to split the filter set. While the algorithm represents a systematic method for decision tree construction, it is disadvantageous to split a filter set using only one header bit per step. In practice, it is desirable to use more bits to generate more subsets in order to reduce the search time. However, evaluating the entropy of multiple bits among 100+ filter bits can be prohibitively time-consuming (i.e., the process complexity is $O(C_m^n)$, where m is the number of filter bits and n is the number of bits chosen per step). In addition, the algorithm does not work directly on most real filter sets because it requires all the filters to be represented as ternary bit strings whereas the port fields are typically specified as ranges.

By contrast, HiCuts [6] and HyperCuts [7] both can generate multiple subsets per step and are practical for handling general filter sets directly. They both take the geometric viewpoint of the problem. A multidimensional space region is cut into some equal-sized subregions at each recursive cutting step. The corresponding subset contains all the filters that overlap the subregion. Locally optimal cutting decisions are made to reduce the size of subsets and to limit the storage expansion. The process is equivalent to making decisions based on a few prefix bits on some header field(s). The major difference between HiCuts and HyperCuts is that HyperCuts works on multiple dimensions (i.e., header fields) simultaneously while HiCuts works on just one dimension at each decision step.

These two algorithms provide better controls on the tradeoff between throughput and storage. They can significantly alter the storage and the throughput by maneuvering the tree fanout and/or the bucket size. However, under a reasonable storage constraint, the throughput of these algorithms is often poor; under a desirable throughput constraint, the storage can become excessively large. There are some other drawbacks to be discussed in the following section. Despite these issues, the decision tree-based algorithms naturally support a pipelined design; hence, they can achieve very high throughput when using a deep pipeline and the parallel bucket matching scheme. Moreover, they allow us to realize simple and efficient implementations using binary encoding techniques. Incremental updates are also easily supported by decision trees. These advantages make this type of algorithms very attractive to real applications. We are motivated to closely examine the root causes of the inefficiency in the decision-tree construction process and to devise new schemes to overcome them.

III. Observations

A good decision tree should have the following properties: The tree consists of as few nodes as possible, and the path from the root to any leaf node is short and balanced. In the previous algorithms, the filter distribution can affect the resulting decision tree significantly.

The first problem, *filter duplication* (i.e., $s_i \cap s_j \neq \emptyset$), is caused by the fact that many filters are weakly specified on some dimensions with wildcards or large ranges. To reduce the tree depth, we prefer to make more cuts at each tree node so that each child node contains fewer filters. However, this also aggravates the duplication problem. As a tradeoff, the previous algorithms typically set a space expansion factor to bound the number of duplicated filters. Without exceeding the threshold, the algorithms make as many cuts as possible per step. This is done purely experimentally without a solid ground for its global impact.

Our simulation profiles show that during the DT construction process, bits from the source or the destination IP address fields are chosen to split the filter set at about 80% of the DT nodes on average. Real-world firewall filter sets typically contain many heavy wildcard filters in these fields; hence they suffer the most from the filter duplication effect. Since HiCuts and HyperCuts can only generate equal-sized cuts, they are not very capable of handling the somewhat contradictory goals of making the tree both thin and short.

The second problem is *skewed filter distribution* (i.e., $|s_i| \gg |s_j|$ for some i and j). Filter distribution in real filter sets is

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

often very skewed. From the geometric view, most filters concentrate in a small region, while a small number of filters distribute across a wide range. For example, in real Access Control List (ACL) filter sets, the filters are fairly specific on the IP address fields, but the distribution is highly skewed. HiCuts and HyperCuts can only make equal-sized cuts per step so the cuts containing more filters need more steps to split, leading to an imbalanced decision tree.

We come to the conclusion that the filter distribution directly affects the DT efficiency. Unfortunately, the cutting strategy of the previous algorithms is not flexible enough to react to this fact. To amend this, we can conceive a simple cutting procedure at a DT node as follows. First, we find the set of optimal cutting points that can balance the distribution of filters among the cuts and minimize the filter duplication effect. Then, we sort and register the cutting points. When a DT node is retrieved during the lookups, we can simply perform a binary search on the point values. The search returns the pointer to the corresponding child node. This method leads to a smaller and shorter decision tree intuitively. One drawback is that the binary searching for the pointer can be slower than the direct indexing in HiCuts and HyperCuts where it takes constant time to get the child pointer. Moreover, storing the cutting points consumes too much storage, which, in turn, would consume too much memory bandwidth for lookups. Ideally, we need to optimize the depth and size of the decision tree as well as the size of the tree node. These insights lead us to propose the *Adaptive Binary Cuttings* (ABC) algorithm with three different flavors.

Adapting to the skewed distribution of filters, the key new idea of the ABC algorithm is to *split the filter set based on the evenness of the filter distribution, rather than the evenness of the cut volumes*. Technically, we have developed three filter-set-splitting strategies so that the algorithm is able to adapt to the filter distribution geometrically or virtually. This additional degree of freedom leads to a balanced decision tree and also reduces the filter duplication effect. We use an efficient binary encoding scheme similar to the one used in the SST algorithm [11], which encodes the space-cutting sequence and can directly map to the bit string of the packet header fields. The encoding scheme makes the new cutting strategies practical and easy to implement in both software and hardware. A toy example is depicted in Fig. 1 to show the flavor of our algorithm, compared to the HiCuts and the HyperCuts algorithms. There are five filters distributed in a 2-D plane. The figure shows a single cutting step. Although our algorithm results in variable-sized cuts, it splits the filters evenly and avoids any filter duplication. On the contrary, HiCuts and HyperCuts both result in regular-sized cuts, but the filter distribution in these cuts is uneven, and some filters are duplicated. Coupled with the efficient encoding, our algorithm is set to outperform the previous algorithms. Note that we cannot use the geometric view to illustrate the third variation of the ABC algorithm as well as Woo's algorithm, but the spirit still holds.

Another important observation concerning the previous algorithms is that their optimization criteria are not strong enough. First, the HiCuts and the HyperCuts algorithms both have to set an expansion factor to control the number of child DT nodes produced per step. Hence each tree node may have a different number of child nodes. For better memory management and

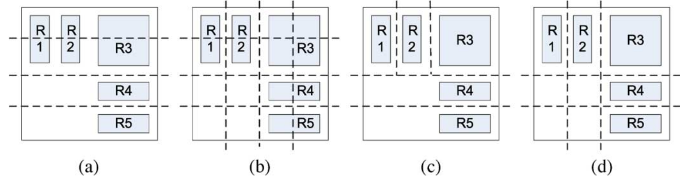


Fig. 1. Cuttings on a DT node for different algorithms. (a) HiCuts. (b) HyperCuts. (c) ABC-I. (d) ABC-II.

easy memory access in real implementations, it is convenient to make every tree node the same size. This causes the nodes with a small fanout to be underutilized. It is also difficult to determine a proper value of the expansion factor. More important, a global expansion factor may not be suitable for all nodes. The ABC algorithm discards the notion of the expansion factor. Since our cutting strategies adapt to the filter distribution, each cut counts, and it does not negatively impact the storage efficiency. Therefore, the ABC algorithm does not need such a parameter. Given the tree node size, we can always fully use the capacity by making as many cuts as possible.

Second, all the previous algorithms stop splitting a set only if the number of filters is smaller than a predefined *bucket size*. Such a criterion cannot guarantee either throughput or storage. The cost to achieve such a goal and its consequence on the quality of the decision tree are left undetermined. Blindly sticking with such a global parameter can worsen the performance in some cases. For some nodes, it might be worth to keep splitting them until each leaf node contains very few filters; for some other nodes, even if the filter set is still relatively large, further node splitting can only do more harm than good. It is difficult to determine a proper global value and to compare the performance of different implementations. The ABC algorithm discards the notion of the bucket size and instead uses a more natural and meaningful approach to make the decision. In our algorithm design, we try to answer the following question: Given a fixed amount of storage, what is the algorithm's achievable maximum throughput? We can also ask the complementary question: In order to achieve a desired throughput, what is the minimum amount of storage required? The second question is harder to answer because, without any prior knowledge, we might set a throughput so high that we can never achieve. On the other hand, the first question is relatively easy. We have known that the minimum storage requirement is simply the storage that is needed to list all the filters. If we store the filters this way, the throughput is solely determined by the number of filters. When more storage is allowed, we can construct a decision tree by intelligently splitting the filter set for higher throughput. Clearly, each decision-tree splitting step will increase the storage monotonically. Our goal is to make the "optimal" decisions that consistently improve the throughput until the given storage is used up. Since the actual performance can be guaranteed only by the worst-case bound, we always try to improve the worst-case throughput performance at each step. This is achieved by evaluating all the current DT branches and the number of filters remaining in each current leaf node, and then choosing the branch that causes the current worst-case throughput to continue working on, as long as the storage budget allows.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Combining the aforementioned elements, the ABC algorithm is not only easy to understand, but also easy to evaluate. In addition to all the above improvements, we also introduce several new refinements to further improve the algorithm's performance.

IV. Algorithm Description

In essence, the decision-making process applies different degrees of freedom for HiCuts, HyperCuts, and Woo's algorithm. HiCuts chooses some number of unexamined prefix bits from only one header field to split the filter set at each step; HyperCuts chooses some number of unexamined prefix bits from multiple header fields at each step; Woo's algorithm chooses any single bit (not necessarily a prefix bit) from any header field per step. With extra flexibility to choose the bits for filter set splitting, the performance of the resulting decision tree can potentially become better. However, the bit-chosen mechanisms have different implications on the storage requirements of the decision tree nodes. For example, if a filter contains D dimensions and L bits (typically $D = 5$ and $L = 104$ for a 5-tuple IP packet filter), HiCuts needs $\lceil \log_2 D \rceil$ bits to encode the cutting dimension at each nonleaf tree node; HyperCuts needs a D-bit bitmap to indicate the cutting dimensions; Woo's algorithm needs $\lceil \log_2 L \rceil$ bits to encode the position of the bit that is used to split the filter set. In addition to this information, HiCuts and HyperCuts also need to encode the number of bits used on the chosen dimension(s). Finally, each tree node needs to store pointers to its child nodes. If all the tree nodes are of the same size, we can store all the sibling child nodes of a tree node in consecutive memory locations; therefore, a base pointer to the first child node is sufficient to address all the child nodes [12]. Note that every additional header bit would double the number of child nodes. We will show that this can significantly affect the overall storage efficiency.

Our new algorithm goes a step further than the previous algorithms in terms of the bit selection strategy. Not only can we choose bits from any dimension and any position to split the filter set, but each resulting subset may consume a different number of filter bits. Basically, the set-splitting decision at each tree node can be represented as one or more full binary *Cutting Shape Trees* (CSTs). We encode each CST with a *Cutting Shape Bitmap* (CSB) that is essentially identical to the SBM in SST [11]. The details will become clear shortly. Right now, let us focus on the high-level idea. The following pseudocode describes the basic decision-tree construction algorithm.

Build_DT

1. Initialize a single-node tree in which the root contains all the filters;
2. while (current storage $\leq$ the predefined storage budget AND
3. some current leaf nodes have >3 filters) {
4. let $S_3 =$ the set of leaf nodes with >3 filters;
5. select $v \in S_3$ which requires the longest time to search a filter in the worst case;
6. split node v to produce the CSTs and the new child DT nodes;
7. }

From the algorithm, we can see that the decision is guided by the memory consumption (line 2). As long as more memory is expendable, the algorithm seeks to use it to improve the lookup throughput. A tree node is not worth splitting further if it contains less than four filters (line 3). Because in such a case if the tree node is split, the resulting child nodes at best contain one or two filters each, but the search path is now one layer deeper. The cost of retrieving and decoding one more tree node is greater than simply performing a linear search on the filters. On the other hand, if a tree node contains four or more filters, it might be worth continuing the splitting process.

The chosen leaf node v identifies the current worst-case searching path (line 5). The path is determined by calculating the maximum cost (i.e., the largest number of bytes) to access the last filter at any leaf node. Clearly, the cost is a function of the leaf node depth, the tree node size, the number of filters in the list, and the filter size. It is easy to find the current worst-case path if we maintain a dynamic sorting data structure such as a heap that uses the current cost as the key. In each loop, we remove the highest path from the heap, split the corresponding leaf node, and then insert the newly generated paths into the heap.

The critical part of the algorithm is how to split a leaf node and produce the CSTs (line 6). Here, we derive three different approaches. We discuss them individually and then compare them together. Before we get into the details, we define an important parameter that is used as a metric for the quality of the DT node cuttings. If at a DT node cutting step, a candidate scheme divides the current filter set into k subsets of size $r_1, r_2, \ldots\ldots, r_k$, we let

$$\mathrm{pref} = \sqrt{\sum_{i=1}^{k} r_i^2}. \tag{1}$$

For example, in Fig. 1(a), the preference value is $\sqrt{3^2+3^2+1^2+1^2} = \sqrt{20}$. Similarly, in Fig. 1(b)–(d), the preference value is $\sqrt{12}$, $\sqrt{5}$, and $\sqrt{5}$, respectively. We consider the best decision as the one that minimizes the preference value. The validity of the metric can be justified heuristically. Note that the preference has the smallest value when the subsets are equal in size. The preference value is also made smaller when the number of duplicated filters is minimized. Hence, the metric seeks to mitigate the two problems mentioned in Section III simultaneously so that a better decision tree can be achieved.

Now we start to describe how a DT node is produced, encoded, and searched.

A. ABC Variation I

1) Producing and Encoding the CST: This variation produces a single CST at each DT node. We map each header field to a space dimension and perform multiple variable-sized cuttings per tree node. The maximum number of cuttings is constrained by the tree node size. The cuttings at a DT node are done step by step. At each step, we choose one of the subregions produced so far and split it into two equal-sized subregions along a certain dimension until we run out of space in the DT node. Each cutting resembles a binary tree node splitting, and each resulting subregion corresponds to a CST leaf node. Therefore, we map the cutting sequence at a DT node to a full binary tree

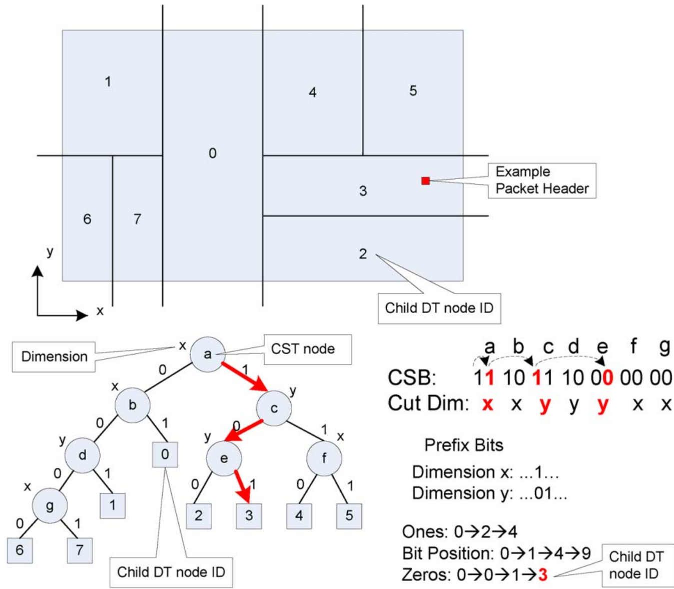


Fig. 2. ABC-I: cuts and lookup on a DT node.

with k leaf nodes. The whole data structure can be envisioned as a tree-in-tree, where each DT node contains a small binary tree that describes the cutting decisions. Fig. 2 shows an example on a 2-D plane where a DT node covers the entire region and the node size allows us to cut the region into eight subregions. Assume we end up with the subregions as shown in the figure. The cutting process can be uniquely described by the binary CST shown in the figure: We cut the region on the x-axis first (node a is labeled with x and is split into two nodes b and c), then we cut the left subregion on the x-axis (node b is labeled with x and is split into two nodes d and 0), and cut the right subregion on the y-axis (node c is labeled with y and is split into two nodes e and f). The process goes on until we generate eight subregions, which map to the eight leaf nodes 0 to 7 in the CST.

All information needed for reconstructing the cutting process is embedded in the CST: the number of subregions k, the cutting sequence, and the cutting shape. The label of the CST node (i.e., the cutting dimension) needs $\lceil \log_2 D \rceil$ bits to encode, where D is the number of header fields involved.

The CST can be efficiently encoded with a bitmap that we name CSB. The encoding scheme is identical to the SBM used in the SST algorithm [11]. We associate each CST node with a bit. The bit value "0" is assigned to the leaf nodes, and "1" is assigned to all the other nodes. The CSB consists of this set of bits, which are listed in breadth-first traversal order. Since the CST root must have two child nodes and the bit associated with it is always "1," we exclude this bit from the CSB and effectively use only $2(k-1)$ bits to encode the CST if there are k leaf nodes. In the example shown in Fig. 2, we encode the CST with the CSB "11 10 11 10 00 00 00." Each "0" in the CSB indicates a subregion (i.e., a new child DT node). The new child DT nodes are indexed incrementally using their order as it appears in the CSB. The first child DT node is given the index 0.

The CST node label that indicates the cutting dimension is also collected in breadth-first traversal order. The label list is shown as the *Cut Dim Vector* (CDV) in Fig. 2. The vector consumes $(k-1)\lceil \log_2 D \rceil$ bits.

For a D-dimensional filter set, it is often the case that some dimensions are never used in a particular CST. We can therefore include a D-bit vector in each DT node to indicate which dimensions are used. Then, if only D' dimensions are used, we need only $D+(k-1)\lceil \log_2 D' \rceil$ bits to encode the cutting dimensions. This optimization can save the memory to allow more cuttings per DT node. For example, if $D=5$, $D'=2$, and $k=8$, this encoding scheme requires just 12 bits in contrast to 21 bits needed before.

After performing the cuttings at a DT node, each subregion represents a child DT node. However, it is possible that some of the child DT nodes contain no filters at all, so it is inefficient to keep a pointer and allocate memory for each of them. Instead, we only keep the child DT nodes with filters. As in the Tree Bitmap algorithm [12], an *Extending Path Bitmap* (EPB) of k bits is used to indicate the presence of the child DT nodes. Bit i of the EPB is set to "1" if the child DT node i is present. Otherwise, the bit is set to "0." When all the child DT nodes are packed together, the pointer to the first child DT node and the EPB are sufficient to address any child DT node.

So far, we have shown how to encode a DT node with up to k child nodes. Now we explain how to come up with the optimal CST at a DT node.

We start from a single node CST that represents the whole region covered by the current DT node. If the current number of CST leaf nodes is less than k (i.e., the number of subregions is less than k), we choose one CST leaf node (subregion) to cut on a specific dimension. We repeat this step until the CST has k leaf nodes. At each step, all possible cuttings on every CST leaf node and on every dimension are evaluated. The CST leaf node and the cutting dimension that can minimize the preference value are chosen to grow the CST.

Formally, assume the current CST has $k' < k$ leaf nodes that divide the DT node filters into k' subsets of size $r_1, r_2, \ldots \ldots r_{k'}$. If we split the node i on the dimension d, r_i is replaced with $r_{i,d,l}$ and $r_{i,d,r}$. Our goal is to find the leaf node i and the dimension d that can minimize the preference value $\mathrm{pref}_{i,d}$. From (1), we have

$$\begin{aligned}\mathrm{pref}_{i,d} &= \sqrt{r_1^2+\ldots+r_{i,d,l}^2+r_{i,d,r}^2+\ldots+r_{k'}^2} \\ &= \sqrt{\mathrm{pref}^2+r_{i,d,l}^2+r_{i,d,r}^2-r_i^2}. \qquad (2)\end{aligned}$$

Hence, in order to find the minimum $\mathrm{pref}_{i,d}$, we only need to evaluate each of the current CST leaf nodes to find the i and d that minimize the value of $r_{i,d,l}^2+r_{i,d,r}^2-r_i^2$.

2) Decoding the CST: The lookup process traverses the decision tree and compares the packet header to the filters stored in the leaf DT node. At each DT node along the path, the CSB needs to be decoded to determine which child DT node to follow.

The CSB decoding algorithm is similar to the SBM decoding algorithm for SST [11]. In CSB, each "0" corresponds to a subregion, and each "1" corresponds to a cutting decision. The goal is to locate the index of the child DT node, for which the corresponding subregion covers the packet header. To describe the algorithm, we index the bits in the CSB with $0, 1, \ldots, 2(k-1)-1$ and index the elements in the CDV with $0, 1, \ldots, (k-2)$, from left to right.

Definition 1: We define $\mathrm{Ones}(s)$ as the number of 1's in the CSB from CSB[0] to CSB[s]. As an exception, $\mathrm{Ones}(0)=0$.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Definition 2: We define $\mathrm{Zeros(s)}$ as the number of 0's in the CSB before $\mathrm{CSB}[s]$.

We perform the following step recursively starting from CSB[0] and CDV[0] until we reach an index t for which $\mathrm{CSB}[t] = 0$, which means a subregion on this DT node has been located.

- Let the current index in CSB be i and the current index in CDV be j. If the value of the next prefix bit from the dimension $\mathrm{CDV}[j]$ is v, the next index i' in CSB is $2 \times \mathrm{Ones}(i) + v$. The next index j' in CDV is $\mathrm{Ones}(i')$.

Once we quit the loop at the index t in CSB, the index of the child DT node is simply $\mathrm{Zeros}(t)$.

A node decoding example is illustrated in Fig. 2 where the region at a DT node is cut into eight subregions. The decoding process needs to figure out the subregion in which the packet drops and then proceeds to the correct child DT node. Since $\mathrm{CDV}[0] = x$, we read the next unexamined header bit from the field x, which is "1." Hence, the next CSB index $i' = 2 \times \mathrm{Ones}(0) + 1 = 1$, and the next CDV index $j' = \mathrm{Ones}(i') = 2$. Since $\mathrm{CSB}[i'] \neq 0$, we need to continue the loop. The next dimension to check is $\mathrm{CDV}[j'] = y$, and the next unexamined bit in dimension y is "0." Hence, the next CSB index $i' = 2 \times \mathrm{Ones}(i') + 0 = 4$ and the next CDV index $j' = \mathrm{Ones}(i') = 4$. Again, $\mathrm{CSB}[i'] \neq 0$, and we need to continue the loop. The next dimension to check is $\mathrm{CDV}[j'] = y$, and the next unexamined bit in dimension y is "1." Hence, the next CSB index is $2 \times \mathrm{Ones}(i') + 1 = 9$. Since $\mathrm{CSB}[9] = 0$, we quit the loop and calculate $\mathrm{Zeros}(9) = 3$ as the child DT node index.

B. ABC Variation II

1) Producing and Encoding the CSTs: In this variation, a DT node is also split on multiple dimensions like the first variation. The difference is that the cuttings can generate up to D separate CSTs, each for one dimension. Fig. 3 shows an example in which a 2-D region at a DT node is cut along both dimensions. The range on the x-dimension is cut into six segments, and a 10-bit CSB is used to encode the corresponding CST: "11 00 10 01 00." Likewise, the range on the y-dimension is cut into four segments, and the corresponding CSB is "01 01 00." Overall $6 \times 4 = 24$ subregions are produced by the two-dimensional cuttings.

To index these subregions, we incrementally label the leaf nodes of each CST starting from zero in breadth-first order. This label, named as *Cutting Label*, uniquely identifies a segment on that dimension. Let the number of cuts on each dimension be $k_1, k_2, \ldots, k_D$. For a particular subregion, let the Cutting Labels on each dimension be $l_1, l_2, \ldots, l_D$. The subregion index is

$$\mathrm{CI} = \sum_{i=1}^{D} \left(l_i \prod_{j=i+1}^{D} k_j \right). \qquad (3)$$

The index of the subregions is shown in Fig. 3. Each subregion, if containing filters, represents a valid child DT node. We use an EPB of $K = \sum_{i=1}^{D} k_i$ bits to indicate the presence of the child DT nodes.

In this variation, there is no need to maintain the cutting dimension at each CST node, hence the same-sized DT node can accommodate larger CSTs. Moreover, the total number of subregions (i.e., the potential child DT nodes) is now determined by the product of the number of leaf nodes of all the CSTs. Assume

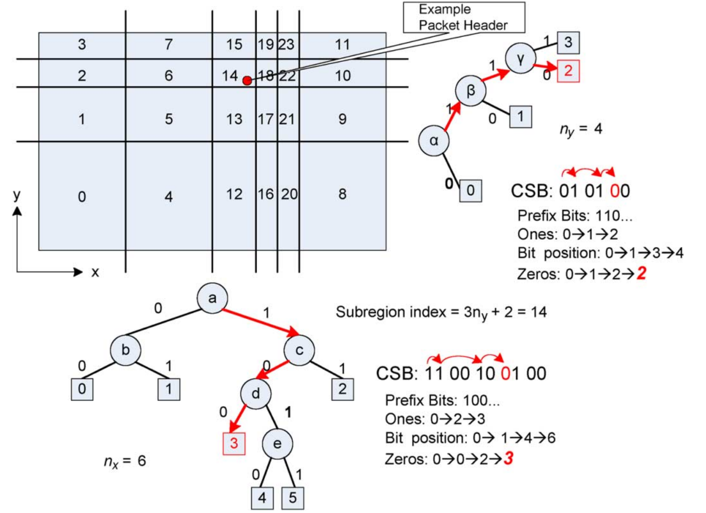


Fig. 3. ABC-II: DT node and decoding example.

we have n CSTs and each CST has k_i leaf nodes. The overall memory consumption of a DT node, excluding the base pointer, is

$$D + 2\sum_{i=1}^{n}(k_i - 1) + \prod_{i=1}^{n} k_i. \qquad (4)$$

The first part is attributed to a D-bit vector that indicates the dimensions used to split the DT node; the second part is attributed to n CSBs that encode the n CSTs; the last part is attributed to the EPB.

To produce the CSTs at a DT node, we start with D single-node CSTs each for a dimension. At each following step, a leaf node on one of the CSTs is chosen to split if the cutting leads to the minimum preference value. The result of the cutting is that a segment on the chosen dimension is split into two equal-sized segments. The process terminates when the bits assigned for the CSTs are used up. At this point, if some CSTs still contain just one node (i.e., they are not split at all), they are excluded from the CSB encoding and the corresponding bits in the D-bit vector are set to "0."

The naive computation of the preference value requires to evaluate the cutting effect of all the current leaf nodes in all the CSTs at each step. To simplify the computation, we perform the similar transformation as in (2). For example, in Fig. 3, we want to evaluate the new preference value if we cut the leftmost segment on the dimension x. Since only the subregions 0–3 will be split, we need only to evaluate their contributions to the resulting preference value.

2) Decoding the CSTs: The lookup process is similar to that in the ABC-I, except now we need to decode multiple CSBs to find the child DT node index. After each DT node decoding, some variable number of prefix bits of the selected packet header fields are examined and used to traverse the decision tree. A decoding example is shown in Fig. 3, where a packet drops in the subregion with the index 14.

C. ABC Variation III

1) Producing and Encoding the CST: This variation produces only a single CST at each DT node. Unlike the other variations where each header field in a filter is deemed as a

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE I
DT NODE BIT CONSUMPTION COMPARISON

Variation	# bits
ABC-I	$(2+\lceil\log_2 D\rceil)(k-1)+k$
ABC-II	$D+\sum_{i=1}^{D} 2(k_i-1)+\sum_{i=1}^{D} k_i$
ABC-III	$(2+\lceil\log_2 L\rceil)(k-1)+k$

dimension, this variation treats each filter as an integral ternary bit string. Akin to Woo's algorithm [8], any bit can be chosen to split the filter set. However, our algorithm is significantly different from Woo's algorithm. First, as we have discussed, the high-level decision-tree construction approaches are different. Second, our algorithm encodes multiple filter bits per DT node. Third, the two algorithms use different preferences to choose the bit for filter-set splitting. Our simulations show our algorithm leads to better performance, which means our preference metric is better than that used in [8].

To produce the CST at a DT node, we start from a single-node CST and keep splitting some leaf node using a bit from the filter string until we run out of the storage space at the DT node. At each step, we evaluate the new preference values for all the leaf nodes if they are split on any filter bit. The leaf node and the filter bit that can minimize the preference value are actually used to grow the CST. The final CST is encoded with a CSB. Each CST node, except the leaf node, must record the filter bit used to split the node, which takes $\lceil\log_2 L\rceil$ bits, where L is the filter length. This information for the entire CST is encoded as a *Cutting Bit Vector* (CBV) similar to the CDV in ABC-I. We use an EPB of k bits to indicate the presence of child DT nodes, where k is the number of leaf nodes in the final CST.

2) Decoding the CST: The CST decoding algorithm is similar to that in the first variation.

D. Comparison

1) DT Node Capacity: Table I summarizes the bits used by the data structures in a nonleaf DT node excluding the base pointer.

The size of nonleaf DT nodes is fixed in real implementations. The decision tree performance is generally better if more cuttings can be done at each DT node. Assume 128 bits are assigned to encode a DT node (excluding the base pointer) and each filter consists of five fields (i.e., $D=5$) and 104 bits (i.e., $L=104$). From Table I, we can derive that ABC-I supports at most 22 cuts per DT node and ABC-III supports at most 13 cuts per DT node. For ABC-II, the maximum number of cuts per DT node is variable, but generally it can produce more cuts per DT node than the other two variations due to the product effect.

2) Implementation Complexity: The second difference has to do with the implementation. Since ABC-I and ABC-III generate a single CST per DT node and the CST can be very tall, the DT node processing latency is typically larger than that for ABC-II, in which all the CSTs can be decoded in parallel. In case pipelines or multiple parallel lookup engines are used to fill the memory bandwidth, ABC-II has smaller system complexity and better performance. However, the preprocessing time of ABC-II is the largest because it requires more computations to produce the CSTs at each DT node.

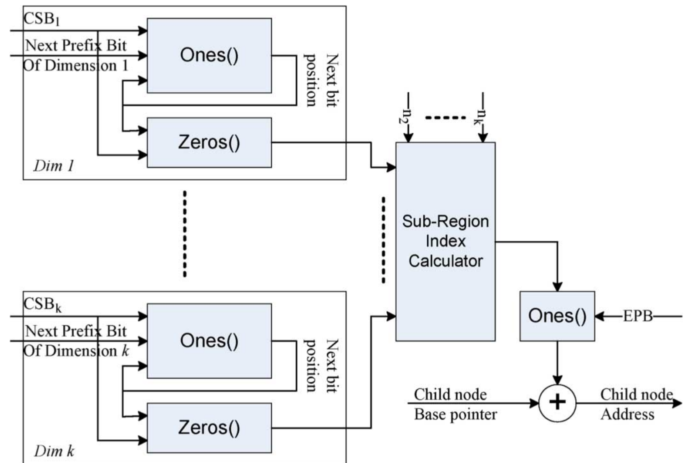


Fig. 4. Decoding the DT node to find the child DT node address.

E. Implementation

The ABC algorithm can be implemented using the state-of-art ASIC/FPGA hardware or NPU-based multithreading software. Multiple lookup engines can work on different packets in parallel to fully utilize the available memory bandwidth. The core component of the lookup engine is the CSB decoding logic. A simple hardware implementation of the CSB decoding uses a sequential circuit to compute the values of $\mathrm{Ones}(\cdot)$, $\mathrm{Zeros}(\cdot)$, and the new bit index iteratively on successive clock ticks. For ABC-I and ABC-III, this takes at most $k-1$ clock ticks. For ABC-II, multiple copies of the circuit work in parallel, each for one CSB. This takes at most $\max_i(k_i)$ clock ticks. We need another one or two clock ticks to calculate the child DT node index and add the offset to the base pointer for the next memory access. Decoding a DT node does not need to access the main memory. Assume we have enough on-chip resource, we can afford a large number of lookup engines. Therefore, the time to decode a DT node only affects the number of lookup engines required. The throughput is merely a function of the available memory bandwidth and the number of memory accesses needed per packet lookup. A block diagram of the DT node decoding circuit for ABC-II is shown in Fig. 4. Note that only one CSB decoding block is required for ABC-I and ABC-III.

The data structure for the algorithm implementation is illustrated in Fig. 5. Note that the nonleaf DT nodes may also hold some filters due to an optimization we adopt in Section V. To save the memory, each filter is only stored once. When a filter must be duplicated, we only duplicate the pointer to the filter because the size of a pointer is much smaller than the size of a filter. We also attach the priority value of each filter to its pointer so that a lookup can determine if a filter needs to be compared without actually reading the filter. A smaller priority value means higher priority. If we have found a matching filter with the priority value i and by reading the filter pointer list, we find a potential matching filter in the list has the priority value $j>i$, we know immediately that the new filter and all the following filters in the list cannot lead to a better match.

Table II shows the DT node encoding scheme of a reference design in which each DT node consumes 16 B, each filter pointer consumes 2 B, and each 5-tuple filter consumes 18 B.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

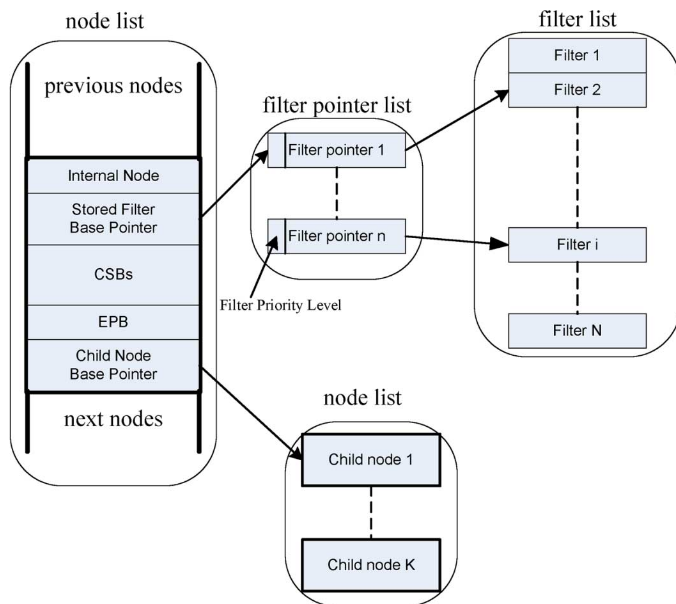


Fig. 5. Data structure of the algorithm implementations.

TABLE II
ABC DT NODE ENCODING SCHEME (# BITS)

	ABC-I	*ABC-II*	*ABC-III*
isLeaf	1	1	1
Cut Dim Bitmap	N/A	5	N/A
CSB(s)	30	*variable*	18
CDV	45	N/A	N/A
CBV	N/A	N/A	63
EPB (i.e. k)	16	*variable*	10
Child Base Pointer	18	18	18
Filter Base Pointer	18	18	18

Note that for ABC-II, 86 bits can be used by the CSBs and the EPB. The assignment of these bits is dynamically determined by the number of CSTs and the size of each CST.

V. ALGORITHM OPTIMIZATIONS

The redundant filter removal optimization introduced in HiCuts [6] and HyperCuts [7] is embedded in the ABC algorithm by default. In this section, we discuss several new algorithm optimizations.

A. Reduce Filters Using a Hash Table

We can use a hash table to handle a portion of the filters so that the number of filters handled by the decision tree is reduced. A hash table $H(i,j)$ uses i prefix bits of the source IP field and j prefix bits of the destination IP field as the key. A filter is hashed into $H(i,j)$ if its source IP prefix specifies more than i bits and its destination IP prefix specifies more than j bits. We test all the possible values of i and j. Assume each hash table bucket can hold T filters. If more than T filters are hashed to a same bucket, only the T filters with higher priority are inserted in the hash table. We choose the hash table that can handle the most filters. The filters in the hash table are removed from the filter set. We evaluate some real filter sets and find that 18%–44% of filters can be removed when $T = 1$, as shown in Fig. 6.

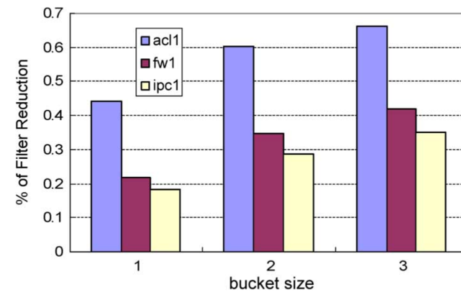


Fig. 6. Effect of filter reduction by using a hash table.

The lookup process needs to search the hash table first. If a matching filter is found, we keep its priority value and continue to search the decision tree. If a lower priority matching is found, the search stops.

B. Filter Partition on the Protocol Field

In all the filter sets we examined, only eight unique protocol values are specified. There are 13% of filters that have a wildcard protocol specification on average.

The cutting does not work well on the protocol field. For example, we need to examine five bits (i.e., produce at least 32 cuts) to differentiate the TCP (0×06) and the ICMP (0×01) protocols. Each cutting unavoidably duplicates all the filters with the wildcard protocol specification. To solve this problem, we build a decision tree for each specified protocol value. Each decision tree handles the filters with the corresponding protocol value as well as all the filters with the wildcard protocol value. The lookup process examines a packet's protocol value first and then search the decision tree dedicated for it. If a dedicated decision tree does not exist, then any decision tree can be searched.

This optimization partitions the filters into a minimum number of subsets and consumes the entire protocol field in just one step. It can reduce the memory consumption and increase the lookup throughput. In addition, it reduces the number of dimensions that need to be considered in the decision trees. For example, in ABC-I, each CST node now needs only two bits rather than three to encode the cutting dimension. Therefore, in the reference design as shown in Table II, a DT node can support 19 child nodes rather than 16. This helps to improve the performance further.

Since ABC-III takes a unified view of the filter bit string, we do not need to apply this optimization to it.

C. Partitioning Filters Based on Duplication Factor

The cutting dimensions and the cutting shape are chosen in favor of the majority filters at a DT node. Throughout the decision tree, some filters suffer more duplications than the others. We profile the number of duplications of each filter for two filter sets when running ABC-II. As shown in Fig. 7, most filters receive zero or very few duplications while a relatively small fraction of filters receive a very large number of duplications. The figure also shows that the higher-priority filters tend to receive fewer duplications than the lower-priority filters.

We identify a filter as a spoiler if it results in excessive duplications. We remove a few top spoilers from the filter set and then build the decision tree on the remaining filters. The

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

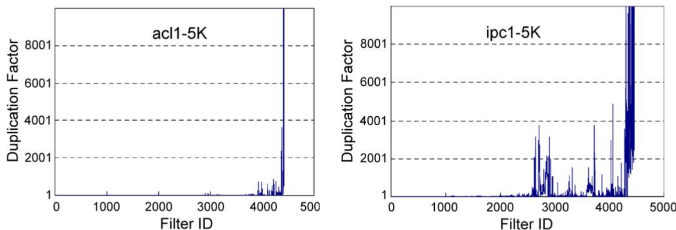


Fig. 7. Filter duplication number distribution.

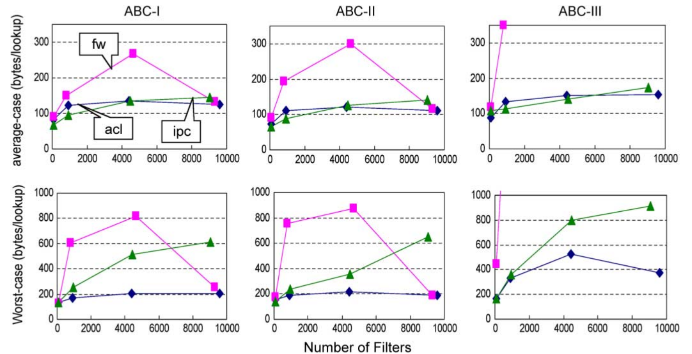


Fig. 8. Algorithm scalability on filter set size.

spoilers can be handled by a small on-chip TCAM. Our simulation shows that this optimization significantly improves the algorithm performance.

D. Holding Filters Internally and Reversing Search Order

The HyperCuts algorithm [7] introduces an optimization called filter pushing, which can reduce the filter duplications but cannot change the tree size and the throughput. We modify this optimization using a forward manner. At a DT node, if a filter would otherwise be duplicated into all the child DT nodes, we can keep it in the current DT node. This modification helps to reduce the tree size as well.

Although efficient in storage, this method worsens the throughput. Fig. 7 shows that the lower-priority filters tend to receive a larger number of duplications. The large number of duplications is largely due to the fact that these filters are less specific, so they effectively overlap with a large number of decision tree nodes. Using the filter pushing, these filters are more likely to be held in nonleaf DT nodes. The low-priority filters also tend to be held close to the DT root. However, the decision tree can only be searched from root to leaf. Even if we find a matching filter at an internal node, we cannot stop. Indeed, we have a good chance to find a better matching filter down to the search path.

We consider to improve the lookup throughput while retaining the gain on the storage. The above observation suggests that we should search the filter lists using the bottom-up order. When we find a stored filter list along the searching path, we do not retrieve the filter list right away. Instead, we push its pointer into a stack. We begin to pop the pointers in the stack and search the filter lists only when we reach a leaf node. Using this order, we search the filters in their natural priority order and can avoid unnecessary memory accesses.

VI. Performance Evaluation

We are concerned with the two most important performance characteristics of the ABC algorithm: storage and lookup throughput. We also discuss the performance of preprocessing and incremental updates. The storage is made up of two parts: the decision tree and the filters. The storage of the decision tree is determined by the number of DT nodes and the size of a DT node. The storage of the filters is determined by the number of original filters and the total number of duplicated filters (recall that each duplicated filter only consumes a pointer). The storage efficiency and scalability are evaluated by the number of bytes consumed per filter. As for the throughput, the depth of a DT branch and the number of filters stored along the branch determine the worst-case performance on the branch. We use the maximum number of bytes retrieved to classify a packet as the worst-case performance measurement criterion. We use a suite of synthetic filter sets generated by ClassBench [13]. For each filter set, we also generate a packet header trace in which the number of packets is ten times of the number of filters. We run the lookup algorithm on these traces to collect the average number of bytes retrieved per packet as the average-case performance measure. The filter sets and the packet traces we used are documented on a public accessible website [14].

A. Comparison of ABC Variations

1) Scalability on Filter Set Size: First, we assume all the aforementioned optimizations are used with the exception of the spoiler filter removal. The hash table bucket size is set to one.

The size of the filter sets ranges from 100 to 10 000 filters. They are synthesized from an ACL seed filter set, an IP Chain (IPC) seed filter set, and a firewall (FW) seed filter set. In the simulation, we set the storage budget to be 100 B per filter. Fig. 8 shows the results.

The algorithm works best on the ACL filter sets. It is interesting to note that memory consumption for the ACL filter set is well below the budget. This means the algorithm has already reached the limit of the decision tree. More storage cannot be exchanged for higher throughput anymore. The algorithm performs the worst for the FW filter sets because these filter sets contain a lot of filters with wildcard specifications.

The worst-case throughput is two to four times worse than the average-case throughput. This is mainly due to the imbalance of the decision tree, although in our algorithm the decision-tree shape has been adapted to the skewness of the filter distribution.

ABC-I and ABC-II show comparable performance. If we optimize the DT node encoding scheme for ABC-I to allow more cuts, ABC-I will outperform ABC-II. The performance of ABC-III is only acceptable for the ACL and IPC filter sets. Although ABC-III is the most flexible variation, it supports the lowest DT node capacity and requires range-to-prefix conversion. These two factors drag down its throughput.

Another interesting point is that the algorithm performs better for the FW filter set with 10 000 filters than for the FW filter sets with 1000–5000 filters. This artificial result is because the ClassBench tool tends to generate more structured and specific filters for larger filter sets.

To interpret the throughput performance, we consider a single 500 MHz $\times$ 36 QDR-III SRAM chip. It provides a

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

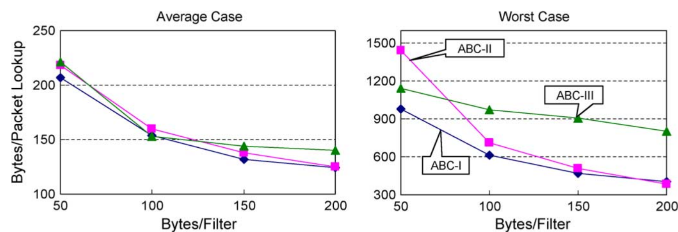


Fig. 9. Tradeoff of storage and throughput.

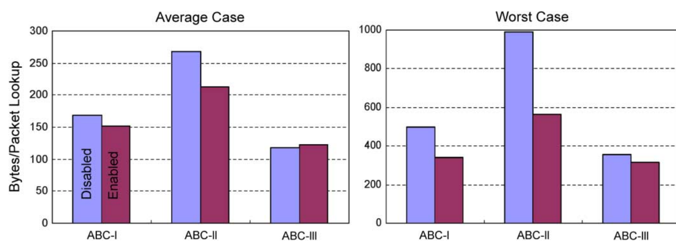


Fig. 10. Effect of filter reduction using a Hash table.

memory bandwidth of 500 MHz $\times$ 9 B $=$ 4.5 GB/s. For the ACL filter set with 10 000 filters, a packet lookup needs to retrieve 125 B on the average. Hence, we can classify $4.5\ \text{GB}/125\ \text{B} = 36$ million packets per second. A fully loaded 10-GbE link can see at most 15 million packets per second. For the average case, the performance of our algorithm is more than sufficient for two 10-GbE links when a single memory device is used. The worst-case performance is two times worse than the average-case performance, so our algorithm is still capable of handling one fully loaded 10-GbE link at line speed.

2) Throughput and Storage Tradeoff: In this simulation, we vary the storage and examine its effect on the lookup throughput. The simulation runs on the synthetic IPC filter set with 10 000 filters. We disable the filter reduction optimization. Fig. 9 shows the results.

When more storage is granted, the lookup performance steadily gets better. All three variations have the similar average-case throughput, but the worst-case throughput differs significantly. ABC-I gives the best overall performance.

3) Sensitivity to Optimizations: Now we examine the algorithm sensitivity to different optimizations. In the simulations, we use the synthetic ACL filter set with 10 000 filters and an allowance of 50 B per filter. We turn on one optimization a time to compare to the baseline algorithm.

Fig. 10 shows the effect of the filter reduction using a Hash table. This optimization significantly improves the worst-case performance (almost 2$\times$ for ABC-II) and moderately improves the average-case performance.

Fig. 11 shows the effect of performing the protocol field lookup first. Note that this optimization is only applied to the first two algorithm variations.

Fig. 12 shows the effect of holding filters internally and reversing the search order. We can see this algorithm optimization only helps to improve the first two variations. It also has the best effect compared to the other optimizations. The reason it does not work for ABC-III is that the decision tree of ABC-III is

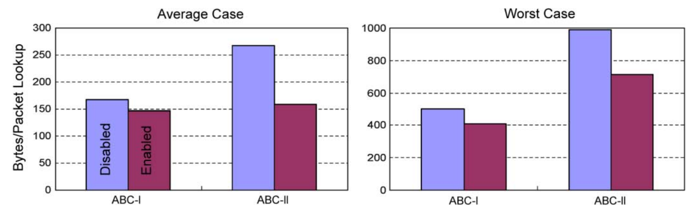


Fig. 11. Effect of looking up on protocol field first.

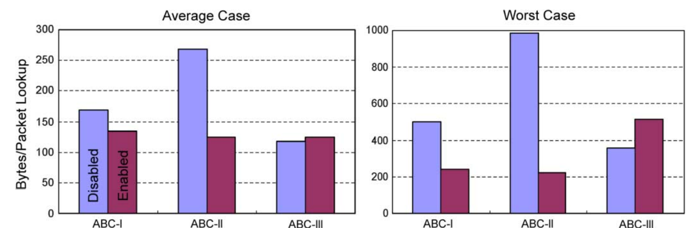


Fig. 12. Effect of holding filters internally and reversing search order.

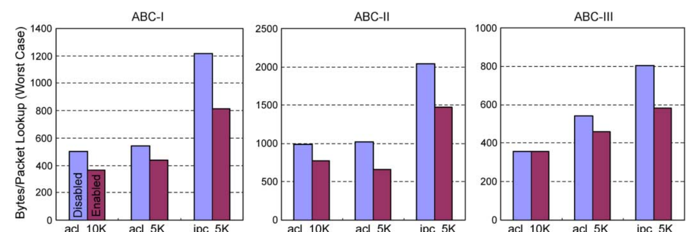


Fig. 13. Effect of removing highly duplicated filters.

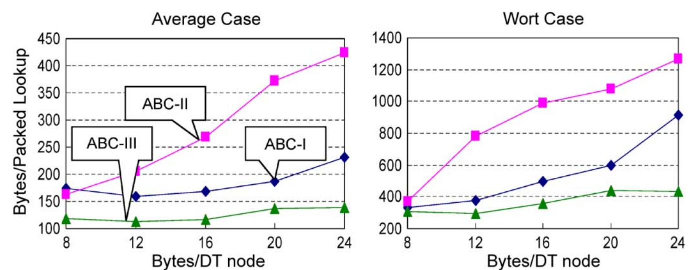


Fig. 14. Effect of changing DT node size.

near optimal and holding filters internally increases the lookup overhead.

Finally, we examine the effect of removing some highly duplicated filters from the filter sets. The duplication statistics are collected from an implementation of the HyperCuts algorithm. Only 3–14 filters (0.1%–0.3%) are removed from the three filter sets. However, Fig. 13 shows significant improvement.

4) Effect of DT Node Capacity: So far, the evaluations are based on our reference design. Now we examine the algorithm performance when different DT node sizes are used. We evaluate five cases with the DT node size of 8, 12, 16, 20, and 24 B. We turn off all the optimizations and allow 50 B per filter. The ACL filter set with 10 000 filters is used for this simulation.

As Fig. 14 shows, in the most cases, increasing the DT node size actually decreases the throughput. This is because under the same storage restriction, larger node size implies fewer DT nodes can be supported. Larger DT node size can improve

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

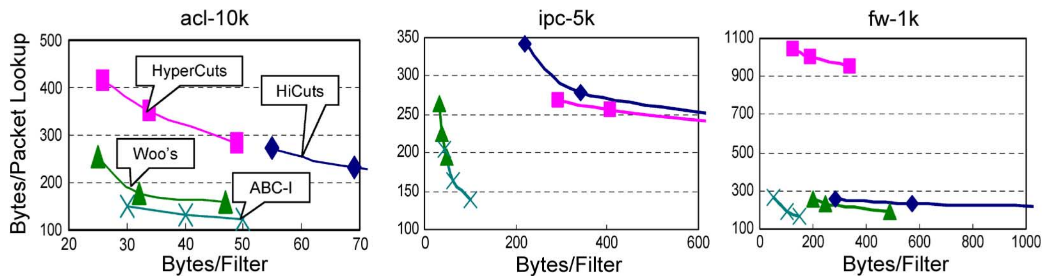


Fig. 15. Compare ABC to other DT-based algorithms.

TABLE III
DT NODE ENCODING SCHEME FOR OTHER ALGORITHMS (# BITS)

	HiCuts	*HyperCuts*	*Woo's*
isLeaf	1	1	1
Cut Dimension(s) (Bitmap)	5	5	N/A
# of prefix bits (Binary Encoded)	3	5*3	N/A
Bit Selection (Binary Encoded)	N/A	N/A	7
Child Base Pointer	18	18	18
Filter Base Pointer	18	18	18
EPB	64	64	2
Total (bits)	109	121	46

the throughput only if we also increase the storage budget accordingly.

B. Comparison to Other DT-Based Algorithms

1) Implementation: It should be clear that for HiCuts [6] and HyperCuts [7], the geometric cutting process is actually identical to the process of examining several prefix bits on some packet header fields in sequence. Since the cuts are regular, no CSB is needed. Each DT node only needs to record which dimension(s) is chosen and how many prefix bits are used. For Woo's algorithm [8], each DT node only needs to record which filter bit is chosen.

We layout the DT node format that also consumes four 32-bit memory words for HiCuts and HyperCuts. A DT node in Woo's algorithm requires only two 32-bit memory words. Just as in the ABC algorithm, we use an EPB and a base pointer to compress the child DT node storage. Note that for HiCuts and HyperCuts, the number of cuttings per DT node doubles for each extra header bit consumed. A DT node supports at most 64 cuts for HiCuts and HyperCuts, which means at most 6 bits can be examined at each DT node. The DT node encoding schemes for the three algorithms are summarized in Table III.

2) Performance Comparison: To make the fair comparison, we apply the same set of algorithm optimizations to all the implementations. Recall that the previous decision-tree-based algorithms terminate the DT construction algorithm only if all the leaf nodes contain fewer filters than a predefined bucket size. Neither storage nor throughput can be fixed before finishing the DT construction. On the contrary, the ABC algorithm allows us to preset one of the performance targets while optimizing the other one. To set the basis for comparison, we run the simulation with different parameter configurations for the algorithms that covers a wide range of storage-throughput tradeoff. Fig. 15 illustrates the results on three different filter sets. The x-axis stands for the storage and the y-axis stands for the lookup throughput. Since ABC-I has the best overall performance, we only show the curve for ABC-I. The performance is getting better when the data point is closer to the bottom left corner. Clearly, the ABC algorithm outperforms all the other algorithms.

3) Preprocessing Time: The preprocessing time differs vastly for the previous algorithms. It depends not only on the filter set size and structure, but also on the parameter settings. For well-structured filter sets and slack parameter settings, the preprocessing can finish within 1 s. However, in many cases, the preprocessing will last much longer and even never end (e.g., the number of overlapped filters in a region is greater than the bucket size). The ideal values can only be determined through multiple trials.

The preprocessing time of the ABC algorithm is linear to the storage budget, and it is independent of the filter set structure. The preprocessing seeks to consume the available memory for the best throughput performance aggressively in a single run. Because the DT node needs to generate and encode the CSTs, it typically takes a few seconds to finish the preprocessing. Fortunately, packet classification filter sets are relatively static and the data structure reconstruction is rare.

C. Incremental Updates

Generally, decision tree does not support incremental updates well due to the filter duplication. To insert a new filter, we may need to push a filter to many leaf nodes. Filter deletion requires a similar amount of work. More importantly, insertion and deletion may lead to suboptimal performance of the data structure, so we have to rebuild the decision tree from scratch at some point. Since the filters can be stored in the nonleaf DT nodes, this can help to reduce the number of duplications. That is to say, if pushing a filter down to the leaf nodes makes too many duplications, we can store it in some internal nodes to limit the duplications. Of course, this should only be done with discretion to avoid degrading the throughput.

VII. CONCLUSION

Decision-tree-based algorithms usually mimic the geometric cutting process, but the decision is conducted in favor of the evenness of the cut size rather than the evenness of the filter distribution. Due to the skewness of the filter distribution found in real filter sets, this approach exaggerates the filter duplication and results in imbalanced decision trees. Woo's algorithm aims to split the filter set more evenly and keep the filter duplication to a minimum. However, it can only produce a binary decision tree

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

with a large tree depth. Moreover, all these algorithms use some indirect criteria to guide the decision-tree construction process, which makes the algorithm evaluation and implementation difficult. The effectiveness of the heuristics is hard to quantify. Both throughput and storage cannot be constrained before the implementation, and the real performance can only be known after experiments. They require a considerable amount of guesswork to fine-tune multiple parameters in several trials for the best result.

We introduce a new degree of freedom to enable variable-sized cuts per decision step in order to even the filter distribution and reduce the filter duplication. This results in a higher-quality decision tree. A simple and compact encoding scheme makes this feasible. The ABC algorithm ensures that all the DT nodes have the same size and are fully utilized. Furthermore, the algorithm applies a natural and performance-guided decision-making process. We preset the storage budget and then look for the best achievable throughput. With just a single knob to tune, our method allows better observability and controllability over the algorithm performance. Based on the similar high-level idea, we derived three variations.

We compare the ABC algorithm to the other decision-tree-based algorithms including HiCuts, HyperCuts, and Woo's algorithm through extensive simulations. The ABC algorithm significantly improves the storage and throughput performance and is scalable to large filter sets. The algorithm implementation is sufficient to sustain the real-time packet classification for 10-GbE lines. The simple implementation and the efficient memory use make the ABC algorithm an ideal alternative to TCAMs and other algorithms in many applications.

REFERENCES

[1] M. Overmars and A. van der Stappen, "Range searching and point location among fat objects," *J. Algor.*, vol. 21, pp. 629–656, 1994.

[2] P. Gupta and N. McKeown, "Packet classification on multiple fields," in *Proc. ACM SIGCOMM*, 1999, pp. 147–160.

[3] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," in *Proc. ACM SIGCOMM*, 1998, pp. 191–202.

[4] T. V. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *Proc. ACM SIGCOMM*, 1998, pp. 203–214.

[5] F. Baboescu and G. Varghese, "Scalable packet classification," in *Proc. ACM SIGCOMM*, 2001, pp. 199–210.

[6] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," in *Proc. IEEE HotI*, 1999, pp. 34–41.

[7] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proc. ACM SIGCOMM*, 2003, pp. 213–224.

[8] T. Y. C. Woo, "A modular approach to packet classification: Algorithms and results," in *Proc. IEEE INFOCOM*, 2000, vol. 3, pp. 1213–1222.

[9] L. Hyafil and R. L. Rivest, "Constructing optimal binary decision trees is NP-complete," *Inf. Process. Lett.*, vol. 5, pp. 15–17, 1976.

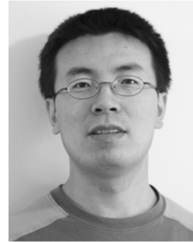
[10] O. J. Murphy and R. L. McCraw, "Designing storage efficient decision trees," *IEEE Trans. Comput.*, vol. 40, no. 3, pp. 315–320, Mar. 1991.

[11] H. Song, J. Turner, and J. Lockwood, "Shape shifting tries for faster IP route lookup," in *Proc. IEEE ICNP*, 2005, pp. 358–367.

[12] W. Eatherton, G. Varghese, and Z. Dittia, "Tree bitmap: Hardware/software IP lookups with incremental updates," *Comput. Commun. Rev.*, vol. 34, no. 2, pp. 97–122, 2004.

[13] D. E. Taylor and J. S. Turner, "ClassBench: A packet classification benchmark," in *Proc. IEEE INFOCOM*, 2005, vol. 3, pp. 2068–2079.

[14] H. Song, "Evaluation of packet classification algorithm," 2007 [Online]. Available: http://www.arl.wustl.edu/~hs1/PClassEval.html



Haoyu Song (M'07) received the B.E. degree in electronics engineering from Tsinghua University, Beijing, China, in 1997, and the M.S. and D.Sc. degrees in computer engineering from Washington University in St. Louis, St. Louis, MO, in 2003 and 2006, respectively.

He is currently a Senior Network Architect with Huawei Technologies, Santa Clara, CA. He was an MTS Researcher with Bell Labs, Alcatel-Lucent, Holmdel, NJ, and a Research Assistant with the Applied Research Laboratory, Washington University. He has published more than 20 peer-reviewed papers and has filed more than 10 patents for his work on network packet processing and network virtualization. His research interests include network virtualization and cloud computing, high-performance networks, algorithms for network packet processing and security, network chip architecture, and ASIC/FPGA design and verification.

Jonathan S. Turner (M'77–SM'88–F'90) received the M.S. and Ph.D. degrees in computer science from Northwestern University, Evanston, IL, in 1979 and 1982, respectively.

He holds the Barbara and Jerome Cox Chair of Computer Science with Washington University in St. Louis, St. Louis, MO, and is Director of the Applied Research Laboratory. He has graduated 21 Ph.D. students and has served as Chair of the Department of Computer Science and Engineering from 1992 to 1997 and 2007 to 2008. He was a Member of Technical Staff with Bell Laboratories, Naperville, IL, from 1977 to 1983, where he provided the technical leadership on an early project seeking to integrate voice and data communication using packet switching. He was Co-Founder and Chief Scientist for Growth Networks, a startup company that developed scalable switching components for Internet routers and ATM switches, before being acquired by Cisco Systems in early 2000. He has been awarded 30 patents for his work on switching systems and has many widely cited publications. His research centers on the design and analysis of high-performance networks, and he has led a series of major systems projects over the years that have demonstrated important innovations in high-performance switching, scalable multicast, extensible routers, and network virtualization.

Prof. Turner is a Fellow of the Association for Computing Machinery (ACM), a member of the National Academy of Engineering, and a member of the Board of the Computing Research Association. He received the Koji Kobayashi Computers and Communications Award from the IEEE in 1994 and the IEEE Millenium Medal in 2000.