

A reconfigurable architecture and associated synthesis methodology for high speed packet classification

Amit Prakash Ramakrishna Kotla Tanmoy Mandal Adnan Aziz
Department of Electrical and Computer Engineering
The University of Texas at Austin
prakash|kotla|tanmoy|adnan@ece.utexas.edu

Abstract—

Packet classification is a computationally intensive task that routers need to perform at line speed in order to implement features such as QoS, access control, and VPNs.

A classification rule-set can be naturally associated with a Boolean function mapping the packet header to an action. Thus it is natural to look at logic synthesis techniques. However there are two major differences from a general logic synthesis problem, (1.) there is a very specific structure in these functions which can be exploited and (2.) these functions change frequently so the target needs to be reconfigurable and updates should be fast. We recently demonstrated that classification on one field, needed for IP forwarding, can be performed at very high speed by mapping the mvBDD representation of the forwarding function to a pipelined array of SRAMs.

Our approach does not immediately generalize to packet classification on multiple fields — the mvBDD for the classification function grows very large. We develop an approach based on partitioned BDDs to overcome this problem. The problem of minimally realizing a Boolean function as a partitioned BDD is NP-complete. We describe a heuristic for building pBDDs that exploits structure derived from classification rule-sets.

The number of memory banks used by the hardware can be reduced by collapsing multiple levels of BDD together. We present an efficient algorithm to obtain optimal grouping of variables that minimizes total amount of memory required under constraint on number of levels in the collapsed BDD.

I. INTRODUCTION

The Internet, as it stands today, offers limited support for rich services, such as quality of service (QoS), security and accounting. Consequently enterprises have been reluctant to commit themselves wholeheartedly to an IP-based infrastructure.

In principle, packet classification could be used to help solve these problems. Packet classification consists of making forwarding decisions and implementing a set of policies specified by the network system administrator that determine the action to be taken for each packet. For example, network service providers could offer guaranteed services to their customers by classifying packets according to which customers they corresponded to. Similarly, denial-of-service attacks could

be monitored using a classifier which identified specific types of requests. However, the computational overhead of packet classifiers in current routers precludes their use.

Packet classification is very similar to the point location problem studied for decades by computational geometers [1]. The best algorithms for this problem either have exponential space complexity and logarithmic time complexity, or vice versa, with the exponential factor being dictated by the dimension of the space (which in the context of packet classification is the number of fields in the header). Currently software solutions use large data structures such as hash tables and tries to reduce time complexity which limits their speed to access time of a DRAM. The hardware solutions mostly use Ternary Content Addressable Memory (TCAM) which are limited in their scaling capabilities because of huge amount of power and area requirements. Due to lack of space, we point to readers to Gupta's thesis [4] for an excellent survey of the field.

Our work is inspired by the insight that packet classification is a special instance of the combinational synthesis problem. We also exploit the special structure in the rule set as they use only prefixes to match the header and most rules do not use more than two prefixes to classify packets. In the synthesis methodology actual latency is not very important but the throughput must be able to match the rate at which packets are arriving. As these rules change we also need a fast way to make small changes. We propose a multivalued BDD based solution that achieves an order of magnitude faster throughput than existing solutions. For convenience we will call multivalued BDDs just BDDs in the rest of the paper.

A. Packet classification — formalization

A popular paradigm for packet classification is to classify packets based on their header. Specifically, in the context of IP, a level-4 classification rule consists of predicate-action pair.

Syntactically, the predicate consists of a sequence of bit-prefixes, one for each field in the header. Semanti-

cally, a packet satisfies a predicate iff for each header field, the packet's value in that field lies in the set defined by the corresponding bit-prefix. The action is encoded by an integer, drawn from some finite interval, and could, for example, denote class-of-service. The rules are assumed to be totally ordered by their priority. For every incoming packet, the classifier is responsible for computing the action corresponding to the highest priority rule that the packet satisfies. (For convenience, we assume the existence of a default rule, which satisfies all packets.)

Every router is required at least to do some basic classification in form of route lookup, which is classification based on just one field, namely destination IP address. For more advanced functions, rules may classify on additional fields, e.g., source IP address, source port, destination port or the protocol field. An example of such rule could be, classify all the packets coming from IP address S and going to destination port 80 (web traffic) as low priority packets.

II. OUR CLASSIFIER

Let the \mathcal{F}_C be the function that maps a sequence of 104 bits (corresponding to the source IP address, destination IP address, source port, destination port, and protocol) to $\log_2 A$ bits encoding one of the A possible actions to be taken. We look at possible implementations of map \mathcal{F}_C .

A. FPGA based synthesis

One of the natural choices to look at is an FPGA, but we found it to be ill suited for our application. We attempted to map logic equations corresponding to a back-bone router's forwarding table to a Xilinx FPGA using the Xilinx logic synthesis tools. The tool ran for a day without succeeding at synthesizing the equations. We then gave the tool a mux-based gate-level netlist implementation derived directly from the BDD representation of mapping function, and told it to perform place-and-route the netlist. In one day it could place-and-route only one of the BDDs (corresponding to the least significant bit of the mapping function), and the delay of the resulting circuit was 85 nanoseconds, which is not competitive with the existing state-of-the-art.

The problem with mapping a large, unstructured set of logic equations to an FPGA is that fitting in so many nodes and their interconnects is again a hard combinatorial optimization problem, especially since there are relatively few long wires in FPGAs. Furthermore, these wires go through many switch boxes inside the FPGA, which adds to the delay.

B. BDD based classification

In [7] we proposed to build special hardware which is designed for fast evaluation of BDDs. Our approach consists of building the BDD for \mathcal{F}_C and then mapping

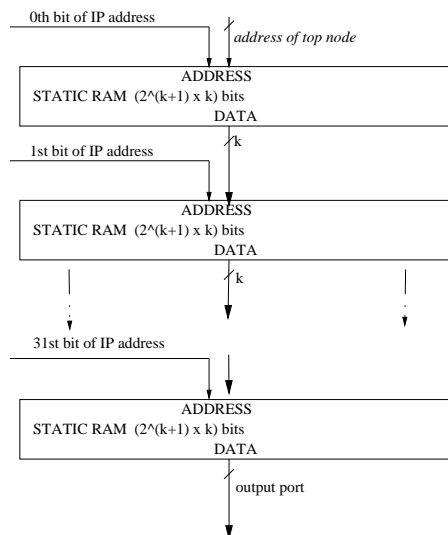


Fig. 1. SRAM implementation.

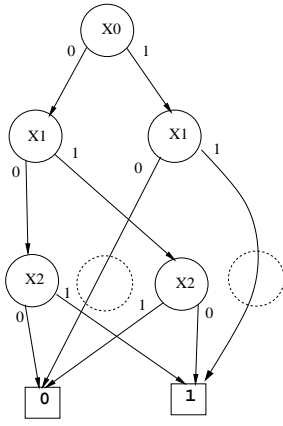
it to a pipeline of 32 SRAMs, numbered from 0 to 31, as in Figure 1. Conceptually, the k -th SRAM holds the BDD nodes for level k ; the data-out lines of the k -th SRAM and the $(k+1)$ -th bit of the input IP address feed the address lines of the $(k+1)$ -th SRAM. (Assume for now that we do not skip levels when the children of a BDD node are equal.) This is illustrated in Figure 2(b).

This approach works very well with classification on one field (for IP forwarding). As memory lookups can be pipelined, the throughput of the system is only limited by the access speed of an SRAM. We could accommodate 57668 rules downloaded from CAIDA [2] in 10 memory banks 16KB after some level compression optimization.

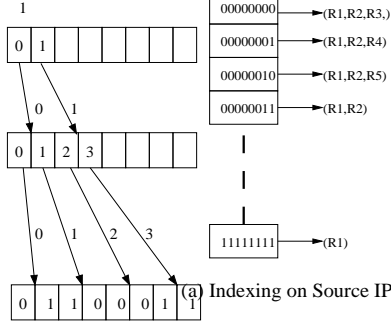
However this approach does not generalize to classification on multiple fields very naturally. In case of classification on one field, it can be shown analytically that the size of BDD representing the classification function is linear in the number rules, as number of cofactors at any level can at most be as many as prefixes. This condition does not hold in case of multiple fields. In fact when rules use multiple fields the size of BDD could grow exponentially in number of fields. We tried building a BDD for just 800 rules and the number of nodes was well beyond a million.

C. Partitioned BDD based classification

The problem of BDD blow up has been encountered in the past in the context of formal verification. Narayan *et al.* [6] proposed one variant of BDD called partitioned BDD (pBDD) proposed to overcome blowup in some cases. They keep a set of window functions whose union is whole input space. A function is represented by a set of BDDs corresponding to the restriction of that function in each window. We also propose to use partitioned BDDs but in a different sense. In our context this amounts to partitioning the rule-set into subsets, and building a separate BDD for each subset. Each subset



(a) BDD



(b)

Fig. 2. (a) BDD for the Boolean logic function $f = x_0 \cdot x_1 + \bar{x}_0 \cdot (\bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2)$. (b) Representation of the BDD in memory.

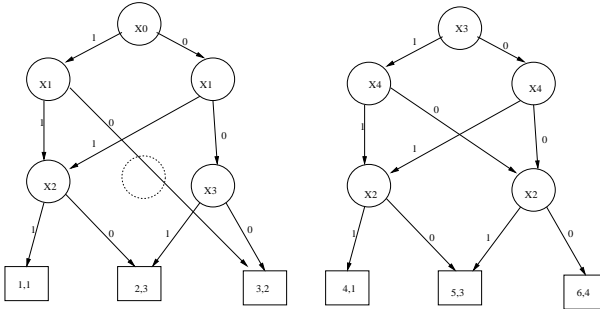


Fig. 3. pBDD representation of a classification function consisting of 2 BDDs. The first value at the terminals is the action; the second its priority.

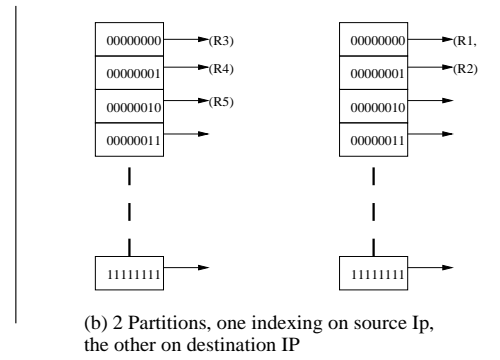
will independently compute an action together with its priority for the incoming packet. A priority encoder will select the highest priority action from all those computed from the individual subsets. This is illustrated in Figure 3.

Narayan *et al.* [6] had no knowledge of the applications their functions were coming from, and consequently proposed a trivial heuristic for constructing partitioned BDDs. Empirically, their heuristic does not help in significantly reducing the BDD size for our application.

We will attack the problem of BDD blowup by partitioning the set of rules into a fixed number of subsets and selecting a different variable ordering for the BDD corresponding to each subset such that the total number of nodes is minimized.

If we have just one rule in each subset then the number of BDD nodes in each subset is bounded by the number of variables, and thus the total number of nodes is proportional to the number of rules. However, we then require as many hardware units as we have rules, and end up with a solution isomorphic to that offered by TCAMs, except that the checking is done in a pipelined fashion.

We actually designed such a TCAM based archi-



(b) 2 Partitions, one indexing on source Ip, the other on destination IP

Fig. 4.

Rule	source IP	Destination IP
R1	*****	00000000
R2	000000**	00000001
R3	00000000	0000****
R4	00000001	110101**
R5	00000010	000****

TABLE I

A RULE-SET ON TWO FIELDS.

ture, all the way down to layout. For 1000 rules, we needed 104000 TCAM cells and 104000 pipeline latches, all of which were clocked in every cycle. Although this architecture yields high throughput, it consumes an enormous amount of power and has no chance to scaling to 10000 rules with current VLSI technology (a TCAM cell is about $20\times$ the size of a SRAM cell; a latch is about $5\times$ the size of an SRAM cell [8]). Thus we need to reduce the number of subsets to a small number, of the order of 10.

1) *Variable ordering and its relationship to partitioning*: Since each BDD node can have at most two children, there can be no more than 2^{k-1} nodes at the k -th level of a BDD (assuming level numbering begins with 1). In particular, there can be at most 2^8 nodes at level 9.

If we traverse a BDD corresponding to a classification function, by taking 8 consecutive 0-branches, we will arrive at a node that will match only those rules that have a 0 or a * in their first 8 positions. By storing this node at location 0 of an array of length 256, and storing the node reached on the path 00000001 at location 1, etc., we will get the structure shown in Figure 4.

In the worst case, there is no node sharing across the BDDs rooted at the nodes stored in the array, and the size of BDD for the subset will be the sum of the sizes of the BDDs rooted at these nodes. If the number of rules that could possibly match the 8-bit prefix corresponding to a node in the array is small, say c , then we are guaranteed that the BDDs rooted at these locations are no wider than 2^c (and possibly much less). Thus our goal is to find a partition that allows us to differentiate between the rules in a subset based on testing a few bits.

Consider the rule-set shown in Table I. As shown in

Figure 4, rule R1 will be present in all BDDs if the variable ordering begins with the source IP bits. In general, for any given field usually, more than half of the rules will have no entry in that field. Therefore, when partitioning we want to group rules that have long prefixes (i.e., less *s) in a common field, and then give that field the top place in the variable ordering for the BDD corresponding to that group. Hence, in the given example if we put R1 and R2 in one subset with the top variables being the Destination IP address bits, and R3, R4 and R5 in another subset with the top variables being the source IP address bits, we will have each of the nodes at level 8 representing only one rule.

The problem of partitioning a set of rules optimally so that the size of pBDD can be minimized, is NP-hard. The decision version of this problem can be shown to be at least as hard as partitioning a graph into triangles which is known to be NP-complete [3]. We tried a greedy algorithm which is suboptimal but gives an effective partitioning that can be computed in a few seconds and used for classification. The algorithm itself is too complicated to describe here. In a simpler version of the algorithm we try all bytes as the top order byte in the header. The one that matches most number of rules so that no matching rule has more than i number of * in that byte and no more than j number of rules collide at one position in the index array, is chosen for a subset and then the matched rules are removed from the set to do the same operation for the next partition. We do that iteratively increasing i and j one at a time till we match all the rules.

D. Architecture

The hardware architecture that we propose to evaluate a pBDD has a pipeline of SRAMs as the basic unit for evaluating the constituent BDDs. (It has been taken directly from [7].) Each unit has as many SRAM banks as the number of levels in the BDD. The nodes of the BDD from the k -th level are stored in the k -th SRAM. The node consists of two pointers in the next SRAM bank for the *else* and *then* branch. The mapping is shown in Figure 1.

Once we have a unit to do the evaluation of one BDD, we just keep as many such units as there are component BDDs in the pBDD. Each BDD unit returns the match of highest priority in that subset, together with its priority as shown in Figure 5. We put dummy default rules in each partition so that at least there is one match.

A priority encoder takes all these results and outputs the final match with highest priority. We will see that 10 subsets suffice to handle up to 10000 rules with reasonably sized BDDs. Thus a small, 4-stage priority encoder suffices.

As different BDD evaluation units will have different variable ordering we also need to permute the inputs for each unit. Permutation in hardware is area intensive, and since most of the time we would want to move a whole header field together in the variable ordering, we

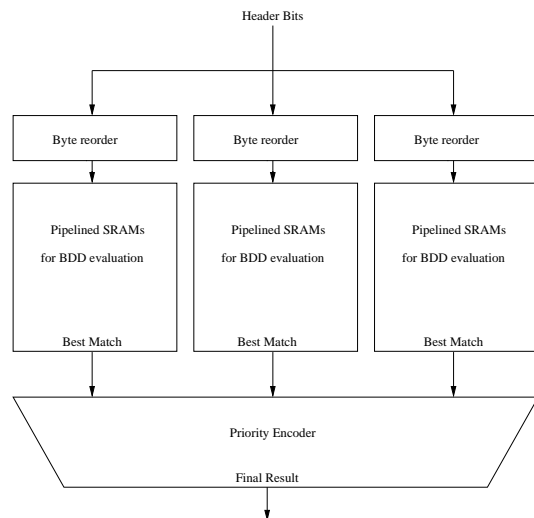


Fig. 5. Architecture for pBDD evaluation

allow the reordering only at the byte level. Since there are 13 bytes in the 5 tuples we are considering we would need an 8 bit wide 13×13 permutation network. This will need 4 stages of 2 input multiplexers.

The power of our architecture comes from the fact that it forms a perfect pipeline — there is no feedback, and stalls are impossible. Hence, we can pipeline the priority encoder and permutation network without any penalty other than increased latency.

III. OPTIMUM COLLAPSING OF BDDs

A generalization of BDD is a Multi-Valued Decision Diagram (MDD) where multiple levels of BDDs can be collapsed into one [9]. Latency and number of memory banks can be reduced in our classification engine by using MDDs instead of BDDs. By collapsing k consecutive levels of a BDD we get rid of $k - 1$ cycles of latency as well as $k - 1$ memory banks but each MDD node in that level becomes 2^k times bigger than a BDD node. This may initially reduce the total memory uses (because of disappearance of levels) but after a threshold it leads to exponential increase in size of MDD. In this section we develop an algorithm for minimizing the total memory requirement given a constraint on number of levels of MDD. McGeer *et. al* [5] have studied MDDs in the context of fast functional simulation. They just mention that dynamic programming can be used for collapsing the BDD levels but do not talk about the trade-off between the memory requirement versus the height of the MDD. The results that we present here is in context of BDDs for packet classification but the algorithm is relevant for any other application where this trade-off is important.

For a given BDD of N variables, the total number of such L groupings will be $\binom{N-1}{L-1}$. So, any algorithm that searches through all possible L level MDDs will have $\Theta(N^L)$ complexity. We present a dynamic programming based algorithm that finds an optimum MDD of height L given a BDD of height N , in polynomial time.

Let $D(i, j, l)$ represent the minimum memory required for MDD formed by collapsing BDD levels i to j into l groups (height of MDD). The recursive formula for calculating $D(i, j, l)$ is

$$D(i, j, l) = \min_{i \leq k < j} [\min_{1 \leq t < l} [D(i, k, t) + D(k+1, j, l-t)]]$$

with terminating case - $D(i, j, 1) =$ Memory required for collapsing all BDD levels $[i, j]$. Essentially this algorithm partitions the levels from i to j into two parts - $[i, k]$ and $(k, j]$. Then, each partition is grouped in such a way that both of them form l groups. In order to compute the minimum sized L variable MDD from an N variable BDD, $D(1, N, L)$ is to be computed. The space complexity of this algorithm is $\mathcal{O}(N \times N \times L) = \mathcal{O}(N^2 L)$, coming from the size of D matrix. Since each entry of the D matrix is computed using the above recursive formula, the time complexity of this algorithm is $\mathcal{O}(N^2 L \times N \times L) = \mathcal{O}(N^3 L^2)$. Since l groups can be formed from BDD variables $[i, j]$ if and only if $j - i + 1 \geq l$, the corresponding entries in the D matrix need not be computed.

function GroupBDDVars (i, l)

```

if (Table( $i, l$ )  $\neq$ )
  return
  Table( $i, l$ ).memory = INFINITY;
  Table( $i, l$ ).best_group = {};
if ( $l == 1$ )
  /* Memory expressed in Bytes */
  Table( $i, l$ ).memory = ( $n_i \times 2^{N-i+1}$ )  $\times$   $\lceil \frac{\lceil \log_2 n_{N+1} \rceil}{8} \rceil$ ;
  Table( $i, l$ ).best_group = { $i, i+1, \dots, N$ };
else if ( $N - i + 1 == l$ )
  /* Memory expressed in Bytes */
  Table( $i, l$ ).memory =  $\sum_{k=i}^N (n_k \times 2 \times \lceil \frac{\lceil \log_2 n_{k+1} \rceil}{8} \rceil)$ ;
  Table( $i, l$ ).best_group = {{ $i$ }, { $i+1$ },  $\dots$ , { $N$ }};
else
  for ( $k = i; k \leq j - l + 1; k = k + 1$ )
    memory = ( $n_i \times 2^{k-i+1}$ )  $\times$   $\lceil \frac{\lceil \log_2 n_{k+1} \rceil}{8} \rceil$ ;
    GroupBDDVars ( $k + 1, l - 1$ );
    memory = memory + Table( $k + 1, l - 1$ ).memory;
  if (memory < Table( $i, l$ ).memory)
    Table( $i, l$ ).memory = memory;
    Table( $i, l$ ).best_group = { { $i, i + 1, \dots, k$ },
      Table( $k + 1, l - 1$ ).best_group};

```

Fig. 6. Algorithm for minimum sized MDD given by the number of levels in the MDD (n_i is the number of nodes for BDD variable i)

We can significantly improve upon the above algorithm. Instead of applying the same algorithm recursively on both partitions, this algorithm merges all the BDD variables in the first partition to form a group and applies the algorithm recursively on the second partition to construct the remaining groups. So, the recursive formula for this algorithm is

$$D(i, j, l) = \min_{i \leq k < j-1+1} [D(i, k, 1) + D(k+1, j, l-1)]$$

with the same terminating case the previous algorithm uses. $D(1, N, L)$ is computed in order to find out minimum sized L variable MDD from an N variable BDD. The space complexity of this algorithm is $\mathcal{O}(N^2 L)$, same as the previous algorithm and its time complexity is $\mathcal{O}(N^2 L * N) = \mathcal{O}(N^3 L)$, which is better than the previous algorithm.

From the above recurrence relation we note that, the first term on the right hand side $D(i, k, 1)$ does not recur and for the second term, the index for the second dimension of D matrix is a constant N . So, we can further optimize the algorithm by removing the second dimension of the D matrix. The recurrence relation after this optimization is

$$D(i, l) = \min_{i \leq k < j-1+1} [M(i, k, 1) + D(k+1, l-1)]$$

$M(i, k, 1) = (n_i \times 2^{k-i+1}) \times \lceil \frac{\lceil \log_2 n_{k+1} \rceil}{8} \rceil$ $M(i, k, 1)$ is computed assuming the memory is byte addressable. In the above recurrence relation, n_i represents the number of nodes for BDD variable i , where BDD variables are numbered 1 to N . The variable n_{N+1} represents the leaf nodes, i.e., cardinality of the range of the function being represented by the BDD. This reduction cuts down the space and time complexity by an order of N . After this modification, the space complexity will be $\mathcal{O}(N \times L)$ and the time complexity will be $\mathcal{O}(N^2 L)$. The dynamic programming based algorithm implementing this recurrence relation is shown in Figure 6.

Hardware implementation requires that the number of locations in each memory be at the 2^m boundary for some $m \geq 0$. We modify memory computation in the algorithm to round off memory size to the nearest integer which can be represented as 2^m . This can easily be taken care of in the algorithm described above by replacing n_i by $2^{\lceil \log n_i \rceil}$ for all i in the algorithm.

IV. RESULTS

A. Performance of partitioned BDDs for packet classification

We created pBDDs for different rule-sets containing 1000 to 10000 rules with 10 subsets. The maximum number of nodes at a particular level determines how much memory we need to allocate for each level.

Total size of the BDD as well as maximum number of nodes at any level scaled approximately linearly in number of rules. For 1000 rules, total number of BDD nodes was 18265 and largest level had 114 nodes while for 10000 rules, the numbers were 113292 and 850 respectively.

From the results on our benchmarks it is more than sufficient to provision for 1024 nodes for any level of the BDD for 10000 rules. In order to reduce the pipeline latency, we propose to read the input 4 bits at a time. This marginally increases the memory requirement but decreases the number of pipeline stages from 104 to 26. We will need 16 pointers, each lg 1024 bits wide, at each

node, i.e., 160 bits of storage at each node. Hence each bank will have to have 20KB of memory. Since we start by taking the top 8 bits as address, we need only 24 levels of SRAM. Since we propose to have 10 such units, we will have 240 20KB units, resulting in needing 4.8 MB of SRAM total. In a modern CMOS process, a 20KB SRAM block can easily deliver one access per nanosecond, and the entire 4.8 MB together with the associated control logic can (just) fit on a single die. If we design the system for 1000 rules with 5 BDD evaluation units and at most 256 nodes per level the memory requirement would be 480KB. Budgeting a generous one nanosecond for wiring delay, our classifier achieves a throughput of 500 Mpps.

B. Collapsing of BDDs

Here we present the results obtained by our dynamic programming algorithm for collapsing multiple levels of BDDs. We applied the algorithm on a BDD that represents classification function corresponding to a rule-set of 57668 classification rules on one dimension, taken from snap-shot of forwarding table of a back-bone router, MAE-WEST. Four more BDDs were used in our study, which were formed by randomly choosing some of the rules from the original forwarding table.

The results were obtained using the algorithm described in Figure 6 for optimizing the size of MDD when represented in hardware. The memory size decreases initially by collapsing the BDD levels as the MDD height decreases from 32 to 25. In this region, by collapsing the BDD we are not only achieving the reduced number of lookups due to height reduction but also reduces the memory required to store MDD. The reason behind the decrease in memory size is that the MDD variables for these MDDs consist of just one or two BDD variables and the memory required for merging two consecutive BDD variables turns out to be *less* than the total memory required to store the BDD nodes for these two variables. In general, merging the BDD levels from i to j (including both) for MDD will yield better memory usage than the BDD if

$$n_i \times 2^{j-i+1} \times \lceil \log_2 n_{j+1} \rceil < \sum_{k=i}^j 2 \times n_k \times \lceil \log_2 n_{k+1} \rceil$$

where n_i is the number of BDD nodes for level i . For grouping two consecutive levels $[i, i+1]$, MDD will require less memory than the BDD if $n_i \times 2^2 < 2 \times n_i + 2 \times n_{i+1} \Rightarrow n_i < n_{i+1}$ (Ignoring the memory widths). After the initial decrease, the curve stabilizes and then starts increasing at exponential rate as the height of the MDD falls below 10. It is interesting to observe that height of the MDD can be reduced by 4 times with just twice the increase in memory, which would translate to reduction in latency by a factor of 4. The height of the MDD can be reduced by 8 times with just 3 times increase in the memory when compared to

the original BDD. However, the trade-off curve entirely depends on the BDD structure hence it is application dependent.

V. CONCLUSION

We have presented an architecture and a synthesis methodology that can be used to do packet classification at 500 million lookups per second. This is an order of magnitude faster than all other solutions. It can scale to as large databases as 10,000 rules.

Though this architecture has specially been designed for packet classification, in general it should be useful for implementing any kind of Boolean function, where throughput is very important and latency is not an issue. We are looking at other applications for this architecture.

We also present efficient algorithm for optimally collapsing a BDD into an MDD to minimize the memory requirement under constraint on number of levels being used in the MDD. This algorithm is not only useful in the context of packet classification but also for minimizing size of MDDs for fast functional simulation or other applications where trade-off between memory and height of MDD is of significance.

REFERENCES

- [1] P. K. Agarwal and J. Erickson. *Advances in Discrete and Computational Geometry*, chapter Geometric range searching and its relatives, pages 1–56. American Mathematical Society, 1999.
- [2] CAIDA. www.caida.org.
- [3] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Co., 1979.
- [4] P. Gupta. *Algorithms for routing lookups and packet classification*. PhD thesis, CS Department, Stanford University, 2000.
- [5] P. McGeer, K. McMillan, A. Saldanha, A. Sangiovanni-Vincentelli, and P. Scaglia. Fast Discrete Function Evaluation using Decision Diagrams. In *International Conference on Computer-Aided Design*, November 1995.
- [6] A. Narayan, J. Jain, M. Fujita, and A. Sangiovanni-Vincentelli. Partitioned ROBDDs—A Compact, Canonical and Efficient Manipulable Representation for Boolean Functions. In *International Conference on Computer-Aided Design*, 1996.
- [7] A. Prakash and A. Aziz. OC-3072 Packet Classification Using BDDs and Pipelined SRAMs. In *Hot Interconnects*, Stanford University, CA, August 2001.
- [8] Jan Rabaey. *Digital Integrated Circuits: A Design Perspective*. Prentice Hall, 1996.
- [9] A. Srinivasan, T. Kam, S. Malik, and R. K. Brayton. Algorithms for discrete function manipulation. In *Proc. Int. Conf. Computer-Aided Design*, pages 92–95, 1990.