

A high-performance architecture and BDD-based synthesis methodology for packet classification

Amit Prakash Ramakrishna Kotla Tanmoy Mandal Adnan Aziz
Department of Electrical and Computer Engineering
The University of Texas at Austin

Abstract

Packet classification is a computationally intensive task that routers need to perform in order to implement basic functions such as next-hop lookup, as well as advanced features such as Quality-of-Service and security.

Formally, a classifier examines each incoming packet, and determines which rules to apply to it. Semantically, the classifier is characterized by a function mapping the packet header to an integer encoding the action to be taken for that packet. The function itself is syntactically presented as a chain of if-then-else statements.

Since the header consists of a fixed number of bits, it is natural to use logic synthesis to implement fast small classifiers in hardware. When doing this, there are two key issues that must be kept in mind: (1.) these functions change over time, so the target architecture needs to be reconfigurable, and (2.) classification functions have a structure which should be exploited.

We show that IP forwarding, which is a special case of classification, can be performed by provably small circuits at very high speed by mapping the BDD representation of the classification function to a cascaded array of lookup-tables. This approach does not immediately carry over to general packet classification — the BDD for the classification function grows very large. We develop a solution based on partitioning to overcome this problem. We prove NP-completeness of optimal partitioning. We describe a heuristic for partition.

The latency introduced by pipelining can be reduced by partially collapsing the BDD. We present an efficient algorithm based on dynamic programming to obtain an optimum grouping of variables that minimizes total amount of memory required for a given number of levels.

1 Introduction

Until recently, Internet routers were little more than general purpose computers connected via a standard bus to simple hardware for transmitting and receiving packets over links. This was because the link bandwidth was low enough for a general purpose processor to implement the entire router functionality. The advent of high-speed optical fiber technology has reversed this situation and today routers and not links are the bottleneck in the Internet [10].

Many of the operations performed by routers are not performance-critical and can still be performed by general purpose computers. Thus, an emerging trend in high-performance router design is to implement the router as a combination of a workstation-class CPU and a set of custom integrated circuits. The processor runs software which performs control functions, e.g., computing best routes and analyzing traffic statistics. The custom chipset is responsible for operations which need to be performed at line speed, e.g., packet encapsulation/decapsulation and longest prefix matching.

We address one of the most computationally intensive tasks performed by routers, namely packet classification. A packet is a sequence of bits, consisting of the header followed by the payload. In the context of the Internet, the header is of fixed length and has different logical fields appearing at fixed offsets. At the bare minimum, a router has to be able to examine the field holding the destination address and search through a forwarding table to decide which interface the packet needs to be transmitted out from. A more sophisticated router may look at other header fields to determine what priority the packet should receive, who should be billed for the packet, whether the packet should be blocked, etc. Mapping a packet header to the action to be taken by router on that packet is referred to as packet classification.

In principle, packet classification could be used to make networks more secure, and to provide Quality-of-Service [9]. For example, network service providers could offer guaranteed services by classifying packets according to

which customers they originated from. Similarly, denial-of-service attacks could be monitored using a classifier which identifies specific types of requests. However, the computational overhead of packet classification in current routers has till now precluded its widespread deployment.

Packet classification is difficult because the time budget for classifying each packet is very small, and the rule-sets are very large. A router switching an aggregate of 1 Terabits per second has to process a packet every 1.2 ns (assuming 100 byte packets); and rule-sets can consist of tens of thousands of rules, and forwarding tables can have as many as one million entries [17].

Our work is based on the insight that packet classification is a special instance of the logic synthesis problem, i.e., the problem of optimally realizing a logic function specified using Boolean equations [14]. Our contributions include an architecture which is reconfigurable and allows for very high-speed operation via pipelining, and heuristic algorithms for mapping classification functions to this architecture.

2 Definitions

2.1 Boolean spaces

Let $X = \{x_1, x_2, \dots, x_n\}$ be a set of Boolean variables, with the variables ordered by their indices. A *literal* over X is either a variable x_i or its complement x_i' . A *cube* over X is a (possibly empty) set of literals over X with the property that if x_i is in the set then x_i' is not, and if x_i' is in the set, then x_i not. A *minterm* over X is a special kind of cube that contains either x or x' for all $x \in X$. Note that a cube containing k literals naturally defines a set of 2^{n-k} minterms.

A *prefix* is a cube, that satisfies the property that if x_i or x_i' appear in the cube then for every $j < i$, either x_j or x_j' must appear in the cube. The *length* of a prefix is its cardinality.

We will find it convenient to represent a cube c on a set of n variables by a string of length n on the alphabet $\Sigma = \{1, 0, *\}$. The i -th symbol of the string S is represented as $S[i]$. Thus $S[i] = 1$ means $x_i \in c$, $S[i] = 0$ means that $x_i' \in c$, and $S[i] = *$ means that $x_i \notin c$ and $x_i' \notin c$.

2.2 IP forwarding

We will first define a special instance of the packet classification problem known as IP forwarding. The IP destination address of a packet is a 32-bit field in the packet header. Let $D = \{d_1, \dots, d_{32}\}$ be a set of Boolean variables. A *forwarding table entry* is defined to be an ordered pair (p, o) , where p is a prefix over D , and o is a positive integer. Logically, o identifies the output port through which packets whose destination IP address lie in p are to be forwarded. A *forwarding table* consists of a set of prefix-integer pairs.

Let (p_1, o_1) and (p_2, o_2) be entries in a forwarding table. The set of minterms defined by p_1 can have a nonempty intersection with the set of minterms defined by p_2 , in which case it is straightforward to see that one must contain the other. In such cases, the Internet Protocol dictates that the the packet is to be transmitted on the output port corresponding to the longer prefix [18].

Let k be the largest integer paired with any prefix in a forwarding table T . The *forwarding function* $\mathcal{F}_T : 2^{32} \mapsto \{1, \dots, k\}$ corresponding to a forwarding table is the function mapping a Boolean vector of length 32 to the port number corresponding to the longest matching prefix in T .

As an example for the case where IP addresses are 2 bits, let $\{(**, 0), (1*, 1), (0*, 2), (11, 3)\}$ be a forwarding table; then the associated forwarding function can be described by $\{(00, 2), (01, 2), (10, 1), (11, 3)\}$.

In summary, the problem of IP forwarding is to quickly compute the output port to transmit a packet on, based on a forwarding table and on the packet's destination IP address. Note that this computation is performed over and over again, so it is acceptable to spend significant pre-processing time building up a data structure from the table that is fast to search. Conversely, since the table changes over time and the changes are incremental, the search structure should be efficiently updatable.

2.3 General classification

A *rule* is an ordered pair consisting of a *predicate* and an integer from a finite set encoding the *action* that the router needs to take on that packet. A *predicate* is a tuple of prefixes, one for each field in the header. A rule is said to be *applicable* to a packet if the predicate of the rule *holds* for the packet header. A predicate holds for a packet if the minterm from each

field in the packet header lies in the corresponding set of minterms from the predicate. For example, the predicate $(0*, 1*)$ holds for a packet with header 0011 (assuming two fields in the header each two bits wide). A *rule-set* consists of a totally ordered set of *rules*; the order relation will sometimes be referred to as the *priority* relation. Multiple rules can be applicable for a packet, in which case the action taken is that of the rule with highest priority.

The following is a concrete example of a rule distributed with the `snort` [16] Intrusion Detection System; we have modified it slightly for readability.

```
alert tcp $EXTERNAL_NET 80 -> $HOME_NET 1054
  (priority:high; mail:yes; action:log;
   msg:"BACKDOOR ACKcmdC trojan scan";)
```

This rule is applicable to packets whose protocol field is `tcp`, destination TCP port is 80, source TCP port is 1054, coming from a machine whose IP address lies in the set of prefixes defined by the macro `$EXTERNAL_NET`, and destined to machines whose IP address lies in the set of prefixes defined by the macro `$HOME_NET`. Each time the classifier encounters a packet satisfying this condition, the router is required to log the message `BACKDOOR ACKcmdC trojan scan`, and E-mail is sent to the network system administrator.

A rule-set naturally engenders a *classification function* which maps the packet header to the action corresponding to the highest priority rule that applies for the packet. In the context of IP, we will be considering five header fields: source IP address, destination IP address, Layer 4 protocol, source port, and destination port [9]. Thus a predicate is a five tuple of prefixes. We will also denote a predicate by a string constructed by concatenating all the strings corresponding to the prefixes in the order in which the fields appear in the header.

2.4 BDDs

The Binary Decision Diagram (BDD) is a data structure that can compactly represent and efficiently manipulate a large class of useful functions of Boolean variables. BDDs have their roots in the decomposition given by the Shannon identity, i.e., $f = x \cdot f_x + x' \cdot f_{x'}$. Recursively applying this decomposition leads to a tree structured representation. A *reduced ordered binary decision diagram* (henceforth BDD) is precisely such a representation, with the added requirements that the variables about which Shannon expansion

takes place occur in a fixed order in the tree, and isomorphic subtrees are merged [2].

Note that in our definition of a BDD, we do not eliminate nodes with sub-children that are equal; this is to make for more efficient hardware implementation, and will be discussed later. Furthermore, we allow the terminals to be arbitrary scalars, rather than requiring them to be Boolean-valued.

3 Synthesis methodology for IP forwarding

We introduce our approach by applying it to a special instance of the classification problem, namely IP forwarding.

Let \mathcal{F}_T be the function that maps the 32 bits of destination IP address to $\log_2 k$ bits encoding the next-hop address based on a forwarding table T with k different next-hop addresses. We now consider ways to compute \mathcal{F}_T . Three properties are desirable of any implementation: (1.) it should be fast in order to keep up with link technologies, (2.) it should be cheap, i.e., use limited hardware resources, and (3.) it should support incremental updates, i.e., as T changes it should be easy to change the implementation of \mathcal{F}_T .

3.1 FPGA based synthesis

FPGAs are a natural candidate for hardware implementation of Boolean functions that change over time. However we found them to be ill suited for our application. We attempted to map logic equations corresponding to a core router's forwarding table to a Xilinx FPGA using the Xilinx FPGA synthesis tool. The tool ran for a day without completing. We then gave the tool a mux-based gate-level netlist implementation derived directly from the BDD representation of the mapping function, and told the tool to perform place-and-route for 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 ns, which is not competitive with existing methods, e.g., Gupta's approach [6] which delivers a lookup in a DRAM access time.

3.2 BDD-based synthesis

We propose a reconfigurable hardware that is designed for fast evaluation of BDDs. Our approach consists of building the BDD for \mathcal{F}_T and then mapping it to a pipeline of 32 memories, numbered from 0 to 31. Each slice is as in Figure 1. Conceptually, the l -th memory holds the BDD nodes for level l ; the data-out lines of the l -th memory and the $(l + 1)$ -th bit of the input IP address feed the address lines of the $(l + 1)$ -th memory. (As stated in Section 2.4 we do not skip levels when the two children of a BDD node are equal.) The hardware is closely related to a top-down walk-based evaluation of the BDD: if we were to hold a 32-bit vector constant at the inputs, and clock the circuit 32 times, the output of the l -th stage would be the pointer to the node the walk would reach at level $l + 1$. The additional bit used to index into the $l + 1$ -th memory is the value of input $l + 1$. This is illustrated with a 3-bit function in Figure 2(b).

The power of this approach comes from the following: (1.) since the BDD is split into multiple memory banks that work independently, each individual memory bank is small and can operate very fast, resulting in a very small cycle time, and (2.) this architecture forms a perfect pipeline without any pipeline hazards so a packet can be processed every cycle time. The costs of pipelining include the area, power, and delay added by the pipe latches, and also increased latency. We add latches after each SRAM stage; for the IP forwarding engine, by our results in Section 3.4.1, an 8-deep pipeline requires $8 \times 16 = 128$ pipe-latches, which is quite acceptable. For a router, latency is acceptable as it simply translates to more buffering. (The delay introduced by the pipeline is of the order of tens of nanoseconds which is insignificant; light travels 3 meters in vacuum in 10 ns.) In Section 3.4 we will show that the number of pipeline stages can be significantly reduced by partially collapsing the BDD without a significant increase in the memory required.

It is known that functions having small sum of products (SOP) representation can have an exponential sized BDDs under all variable ordering. However the number of nodes in a trie data-structure representing a forwarding table grows linearly in the size of forwarding table [5]. Since BDDs have an added advantage of node sharing, the number of nodes in a BDD representing a forwarding table T also grows as $O(|T|)$. Since the number of bits needed to store a pointer in a BDD node grows as $\log_2 T$, the overall memory requirement grows as $O(|T| \log |T|)$ [21].

3.2.1 Results

The upper bound stated above does not account for any sharing of nodes in the BDD. In practice we have observed that sizes of BDDs arising from forwarding tables are significantly smaller than the upper bound by an order of magnitude. We downloaded a snapshot of the forwarding table of one of the big core routers, MAE-WEST with 57668 prefixes. The BDD corresponding to the forwarding table had 45063 nodes, the largest number of nodes at any level was 6803. The distribution for BDD nodes at different levels is shown in Figure 5. The time taken to build the BDD was under 10 seconds on a 450MHz Ghz Pentium-III running Linux.

We experimented with random subsets of the forwarding table to get an idea of how the BDD size scales with number of prefixes. Figure 4 shows how BDD size varied with number of prefixes. The growth is sublinear. We also generated random set of prefixes that follow the same length distribution but the resulting BDD had very little node sharing and consequently the BDD was large. Thus the structure in the set of prefixes is important.

We now argue that it should be feasible to build a chip that should be able to compute 500 million forwarding decision per second. Specifically, if we provision enough memory to hold 16000 nodes at any level (which is more than twice the number of nodes seen at the maximum level), we need less than 64KB of memory for each bank. An SRAM holding 64KB with an access time of 1 ns can be easily made using current technology. If we provision 1 ns for interconnect delay, this system can process 500 million packets per second. In contrast, the fastest existing forwarding engines are based on CAMs and are limited to about 100 million lookups per second. (Details on CAM-based classification chips can be found in product literature from Cypress Semiconductor.)

3.3 Updates

As the forwarding table changes over time, we need a mechanism to modify the nodes stored in memory. Usually a small fraction of prefixes are added or deleted to the routing table roughly every 30 seconds. For example the forwarding table of a big core router, containing forwarding entries of the order of 50000, may change by at most 1000 entries every 30 seconds.

One way of supporting updates is to have two copies of the forwarding engine, and alternate between using one for forwarding, and the other for

updating. As we can build the BDD for the forwarding function in a few seconds, this approach is feasible. However, it takes two times the hardware of a single forwarding engine.

We now estimate the cost of performing incremental updates on a single forwarding engine. This can be done as follows: a separate control processor is used to compute the new BDD off-line whenever the updates arrive. It determines which nodes need to be created and which nodes can be freed, makes the memory management decisions and then sends the write commands to the memory array. Since only a small fraction of forwarding entries change, only a small number of nodes in the BDD change. Consequently, the write commands can be transmitted on a relatively slow link, and the forwarding engine is stalled for a relatively small amount of time. For example, let the amount of memory for each bank be 64KB and the write latency of the memories be 2 ns. Even if we overwrite all of the words in the memory every 30 seconds, we would be writing for $64K \cdot 2$ ns, i.e., 0.128 ms out of the 30 seconds. Thus the writes cause the engine to stall for less than 0.01% of the time. The bandwidth between the control processor and each bank will be of the order of $64K \times 8 \times 30 = 1746$ bits/second, which is tiny relative to modern serial link technologies.

3.4 Optimum collapsing of BDDs

Collapsing multiple consecutive levels of BDDs results in a Multi-Valued Decision Diagram (MDD) [20]. We can reduce latency and number of memory banks in our classification engine by using MDDs instead of BDDs; by collapsing k consecutive levels of a BDD we remove $k - 1$ 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 the MDD. (In the extreme case, collapsing all levels results in a direct address table [3].) In this section we develop an algorithm for minimizing the total memory requirement given a constraint on number of levels of the MDD. McGeer *et al.* [12] have studied MDDs in the context of fast functional simulation. They 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 are in the 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 ordered variables, the total number of groupings of the variables into L contiguous blocks will be $\binom{N-1}{L-1}$. Thus looking through all possible groupings will have an exponential time complexity. We present a polynomial time algorithm based on dynamic programming that finds an optimum MDD of height L given a BDD of height N .

Let $D(i, l)$ represent the minimum memory required for MDD formed by collapsing BDD levels i to N into l groups (height of MDD). The function $D(i, l)$ satisfies the recurrence relation,

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

where $M(i, k)$ is the memory required for collapsing all the levels in $[i, k]$. Assuming the memory is byte addressable, $M(i, k) = (n_i \cdot 2^{k-i+1}) \cdot \lceil \frac{\lceil \log_2 n_{k+1} \rceil}{8} \rceil$ where, n_i represents the number of nodes for BDD variable i ; the 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. Using the above recurrence relation, we develop a dynamic programming based algorithm to compute $D(1, L)$. We construct an $N \times L$ array to store the the computed values of $D(i, j)$. A recursive call to the subroutine is made only if the value is not already present in the table. Since the table has $N \cdot L$ entries, at most $N \cdot L$ calls can be made to the recursive subroutine. Hence the resulting algorithm has the space complexity of $O(NL)$ and the time complexity of $O(N^2L)$.

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.

3.4.1 Results

We applied the collapsing algorithm on a BDD that represents the classification function corresponding to a rule-set of 57668 classification rules on one dimension, taken from a snapshot of the forwarding table of a core router. The distribution of number of BDD nodes at different levels in the BDD representing the classification function is shown in Figure 5. We used four more BDDs in our study, which were formed by choosing random subsets of

the original forwarding table.

Figure 6 shows the minimum memory required to represent MDD with varying number of levels in the MDD for all the five BDDs (BDD-I corresponds to the original classification table). It took under three minutes on a 450 MHz Pentium-II to compute all the points on the trade-off curve for BDD-I.

The memory size decreases initially by collapsing the BDD levels as the MDD height decreases from 32 to 25. The reason behind the decrease in memory size is that the MDD variables for these MDDs consist of just 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.

After the initial decrease, the curve flattens and then starts increasing rapidly as the height of the MDD falls below 10. We observe that the height of the MDD can be reduced by a factor of four with just twice the memory; it can be reduced by a factor of eight with just three times the memory.

4 Classification on multiple fields

The linear upper bound on BDD size for IP forwarding does not hold for the problem of classification on multiple fields. We will show this by presenting an example of a rule-set of size r over d fields that needs $\Omega((r/d)^{d-1})$ BDD nodes.

Example 1 Number the fields 0 through $d-1$. Consider a rule-set such that the i -th rule has a nontrivial condition only on the $(i \bmod d)$ -th field and the d -th field. In this set there are exactly r/d rules that have a condition on the i -th field for $0 \leq i < d-1$. We choose the set of conditions on any field is such that exactly one condition will be true for any value assigned to that field (We assume that each field is at least $\log_2 r$ bits wide, thus it is possible to choose such a set of conditions).

Now if we construct the BDD for this rule-set under the variable ordering in which the bits corresponding to the i -th field come immediately after the bits corresponding to the $(i-1)$ -th field, we will argue that there are at least $(r/d)^{d-1}$ nodes in the BDD at the level where variable corresponding to the d -th fields start. A node at this level must encode which of the r/d conditions apply at each of the $d-1$ fields, since any of the matching rules

may be eliminated because of the conditions on the d -th field. As there are $(r/d)^{d-1}$ possibilities, there must be as many BDD nodes to distinguish them.

The trie data-structure is similar to a BDD except that there is no node sharing in a trie. An upper bound of r^d nodes for tries is well known [5]. It is straightforward to see that this bound holds for BDDs as well. Thus under most general case there is no large asymptotic advantage of using BDDs.

Currently rule-sets with 10,000 rules are being used and there is a demand for supporting even bigger sets. For $d = 5$ we will be looking at BDDs with potentially, more than $2000^4 = 1.6 \times 10^{10}$ nodes. Thus our approach of mapping the BDD for the classification function to an array of memories is not guaranteed to work directly because of BDD blowup. One might hope that practical rule-sets may have some hidden structure such that the resulting BDDs are small but our experiments show otherwise: we tried building a BDD for a rule-set containing just 800 rules and found the BDD size reached well over a million nodes.

4.1 Partitioning the rule-set

If we look at Example 1 carefully, it is clear that the rule-set could be partitioned into d subsets each containing rules that have condition on only two fields. If we built a BDD for each of these subsets the resulting BDDs would be small as there are only 2 fields and r/d rules. The sum of BDD sizes then would be at most $O(d \cdot (r/d)^2) = O(r^2/d)$. Thus if we have d memory-arrays for evaluating each of these BDDs and then we merge the outputs of these arrays using a priority encoder to determine the action, that would be much more efficient solution than using a single BDD.

Taking this idea to the limit, any rule-set can be partitioned such that every subset has only one rule. As the size of a BDD representing a single rule can be bounded by a constant, the cumulative size of BDDs will be linear in 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 ternary content addressable memories (TCAMs). The problem with this solution is that even though we are using small memories, it is partitioned into such small banks that peripheral logic around the memory starts dominating. We tried to design such a hardware but the result was not competitive as it consumed large amounts of power and area; details are given Section 5.1. Hence we would like to use a small, fixed number of memory-arrays and

consequently we would like to be able to partition the rule-set into a small number of subsets such that the cumulative size of resulting BDDs is small.

Clearly, even if we partition a rule-set of size r randomly into p equal sized subsets, the resulting BDD size can be upper bounded by $(r/p)^d \cdot p = r^d/p^{d-1}$. Thus if $d = 5$ and we use 10 subsets, our worst case goes down by a factor of 1000. However we know that if there is certain structure in the rule-set (as exhibited in Example 1) we can do exponentially better by partitioning.

One aspect of BDDs that we have ignored so far is variable ordering. The size of the BDD for a function can depend critically upon the variable ordering. It is possible that one variable ordering gives a linear sized BDD while another variable ordering results in a BDD having an exponential number of nodes. If two functions require different variable orderings in order for their BDD representation to be small, there may be no ordering under which their union is reasonably sized [21]. In our context it is possible to have two sets of rules that have compact BDD representations individually but their union has a large BDD. Thus we may get significant compaction by partitioning because of the freedom to choose different variable ordering for each subset.

The above reasoning suggests that if we randomly partition the rule-set into equal sized sets and choose the variable ordering best for each subset, we will get a set of small BDDs. However we can do much better than that by grouping rules that have nontrivial (not always true) conditions on similar fields such that for each subset there exists a variable ordering that can be used to get very small BDDs. Our objective is to find a partition of a given rule-set, and find a good variable ordering for each subset so that the BDD representation of each subset is compact. Each BDD will return the highest priority matching rule for its rule-set and then a priority encoder will be used to obtain the highest priority matching. If we do this (1.) the total hardware cost will go down as we will use a small amount of memory, and (2.) we can use much faster memories as each memory bank will be small.

However there are two issues that must be addressed. First, the number of subsets in the partition must be small. Second, we cannot use arbitrary variable ordering as that requires implementing a costly interconnection network to feed in the inputs for each pipeline.

4.2 Variable ordering and its relationship to partitioning

The problem of optimally partitioning the rule-set so that the resulting BDDs are compact under corresponding optimum variable orderings has two hard combinatorial problems embedded in it that need to be solved simultaneously: optimum partitioning and optimum variable ordering for each subset of rules. The interplay between these two makes this problem extremely difficult.

Our approach takes advantage of the following observation. For a given rule-set if there exists a small *index set* I consisting of integers between 0 and $L-1$ such that for any pair of rules in the rule-set there is at least one integer $i \in I$ such that the predicates corresponding to the rules differ at the i -th bit, then the knowledge of header bits indexed by I will allow us to eliminate all but one rule from consideration. Thus if we use the bits indexed by I as the top variables in the BDD ordering then any node at level $|I|$ will be matching only one rule, so the BDD rooted at that node will be of constant size. At the $|I|$ -th level of a BDD there can be at most $2^{|I|}$ nodes so if we can bound I within a small constant T then we will have a small BDD. Thus our approach consists of partitioning the rule-set into small subsets such that each of them has such a small index set.

As an example consider the predicates shown in Table 1. The predicates have two parts, one corresponding to the source IP address bits and another corresponding to the destination IP address bits. As shown in Figure 7(b), if we put R3, R4 and R5 in one subset and R1 and R2 in another then 8 bits of source IP address can be used as index bits for first subset and 8 bits of destination IP address can be used as index bits for second subset. Figure 7(a) shows if we do not partition the rule-set then even after reading first 8 bits of source IP address the BDD will still have significantly more rules in consideration. In general, since different rules use different fields to classify it is not possible to obtain a small index set that can work for all the rules. However, by partitioning we can separate rules that favor different index sets.

4.3 Partitioning Algorithm

Ideally we would like our partitioning algorithm to return a small number of subsets of the rule-set, with each set having a small number of index bits such that by just examining these bits from a packet header we could eliminate

all but one rule from any subset as a potential match.

The index set needs to have a small cardinality for this approach to be feasible. Suppose our goal is for this set to have no more than T elements. Formally, we would like a partition of R (call it $\{P_1, P_2, \dots, P_p\}$), and a set of index positions I_i for each P_i such that,

1. for all i , $|I_i| < T$,
2. for any pair of rules in P_i with corresponding predicate strings s_j and s_k , there is at least one index position $l \in I_i$ such that $s_j[l] \neq s_k[l]$; furthermore, $s_j[l] \neq *$ and $s_k[l] \neq *$.

In Appendix A we show that the decision version of this problem is NP-complete. Hence we can only hope for an approximation to the optimum solution.

4.3.1 Heuristic partitioning

In the idealized version of the partitioning problem stated above, the sets of index positions were assumed to be arbitrary. A logic circuit which can be programmed to extract a specific subset of the input bits can be built using a permutation network [11]. In principle, it is possible to build a network capable of realizing any permutation of N -bit inputs using $O(N \log N)$ primitive gates; however, the layout area has a much worse lower bound — $\Omega(N^2 / \log N)$.

In order to reduce the complexity of the hardware for the permutation network, it is natural to consider networks which can realize a subset of all permutations. The different fields in the packet header lie on byte-boundaries. Heuristically, we will not lose many high-quality partitions by requiring that the input vector can be manipulated at the byte level only. Furthermore, by doing so, not only do we get a significantly smaller interconnection network, but we also dramatically reduce the the number of possibilities for the index set.

We can also allow violations of the restriction that only one rule should remain as a potential match after examining index bits, and instead try to heuristically minimize the number of violations, i.e., for a particular assignment to index bits there should be a small number of predicates that agree over the index set.

For each set in the partition define 2^T buckets, each corresponding to a particular assignment to the index bits. In any set from the partition a rule is put in all the buckets that have an assignment matching the rule. If a rule has c *s in the index bits then that rule will be present in 2^c buckets. We would like to pick the rules for a set with small c such that average occupancy of buckets is small.

4.3.2 *Greedy_Selection*

We now describe a rule-set partitioning heuristic based on the principles stated above. *Greedy_Selection* takes three user-defined parameters — K , C , and α . The parameter K is the maximum number of rules that are allowed in a bucket; C is the maximum number of *s a rule can have in index bits. The parameter α is a real number; it will be slightly smaller than 1, and its role will be explained later.

Since we rearrange the input at the byte level only, there are much fewer sets of index bits to choose from. The procedure tries all possible index bits for a set in the partition and selects as many rules as it can under the C and K constraints, and decides on the index bits that yield the largest set. The selected rules are removed from the rule-set and the procedure is repeated with the remaining rules.

Once this procedure has gone through p iterations, we will have p disjoint subsets of the rule-set. However, there may still be some rules remaining unselected. We repeat the above procedure by relaxing either C or K by one, until α of the rules have been assigned to a subset; remaining rules are distributed among the subsets based on where they occupy least number of buckets.

Iterating through the *Greedy_Selection* procedure gives an optimal solution because it gives a solution with minimal value of K and C . If we can bound K to a small constant, each node at the T -th level of each BDD will have to represent a BDD for at most K rules so it can have at most 2^K nodes. Hence each rule can contribute at most $2^K/K$ nodes. So there can be at most $N \cdot (2^K/K)$ nodes. If K is a constant then the data-structure grows linearly in N . A small value of C makes sure that each rule is contributing to only a small number of BDDs rooted at T -th level. This helps lower the value of K .

4.4 Architecture

The hardware architecture that we propose has a pipeline of SRAMs as the basic unit for evaluating the constituent BDDs (exactly as in Section 3). 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.

We just build as many such units as there are components in the partition. Each BDD unit returns the match of highest priority in that subset, together with its priority as shown in Figure 8. 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. As stated in Section 4.3.1, permutation in hardware is area-intensive, and we decided to allow re-ordering of byte-size blocks only. 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 again comes from the fact that it forms a perfect pipeline.

4.5 Results

We partitioned different synthetic rule-sets containing 1000 to 10000 rules with 10 subsets to determine how the BDD size scaled with number of rules. Figure 9 shows how the total number of BDD nodes increase with number of rules; Figure 10 shows how the number of nodes at the largest level scales with number of rules. The maximum number of nodes at a particular level is an important metric because it determines how much memory we need to allocate for each level. As it can be seen from the figures, the total number of nodes as well as the maximum number of nodes at any level scaled roughly linearly in number of rules. It took of the order of a few minutes to partition rule-sets containing 10000 rules on a Linux machine based on a Pentium-II running at 450 MHz. The total time taken to build the BDDs for each subset was less than a second.

For 1000 rules; the total number of BDD nodes was 18265 and largest level had 114 nodes, while for 10000 rules, the numbers were 113292 and 850 respectively. According to 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 $2^4 = 16$ pointers, each $\lg 1024 = 10$ bits wide, at each node, i.e., 160 bits of storage at each node. Hence each bank needs 20KB of memory. Since we start by taking the top 8 bits as address, we need only 24 levels of lookup tables. Since we propose to have 10 such units, we will have 240 of 20KB units, resulting in needing 4.8 MB of memory total.

In a modern CMOS process, a 20KB static RAM can easily deliver one access per nanosecond, and the entire 4.8 MB of SRAM together with the associated control logic can fit on a single die. For 1000 rules allocating 256 nodes per level, the memory requirement would be 480KB. Budgeting a (generous) 1 ns for wiring delay, our classifier achieves a throughput of 500 million packets per second.

4.6 Updates

As small changes are made to the rule-set it is very likely that choice of index bits does not change. Thus we propose to keep the index bits same for small updates and add the new rules to one of the subsets such that it can be accommodated in the corresponding memory-array and increase in the cumulative size of BDDs is minimal. If an old rule is deleted, it is simply deleted from the corresponding subset. The partitioning algorithm will be run in the background periodically to compute better partitions. Even though it takes of the order of minutes to run the algorithm, since it is not on the critical path, it does not affect regular updates.

In order to update the component BDDs we use the same mechanism as described in the Section 3.3.

5 Previous work

Prior approaches to classification can be categorized as belonging in one of two extreme categories: (1.) an incoming packet is fed to custom hardware,

based on content addressable memories, which concurrently checks all rules for applicability and returns the action of the highest priority applicable rule; (2.) a (possibly compressed) tree-like structure is stored in memory, and traversed based on the bits in the incoming packet’s header. Both these approaches suffer from severe limitations: the former, though it exploits parallelism, uses a large amount of hardware resources; the latter does not use the parallelism available, and can require huge amounts of memory in the worst case.

5.1 Content Addressable Memories

Logically, a content addressable memory (CAM) consists of a set of key-value entries. When a key is passed into the CAM, it returns the value corresponding to an entry whose key matches the one passed in. The compare operation takes place in parallel, and hence lookups are very fast. The stored keys can hold don’t care values for specific bits, in which case the CAM is called a Ternary CAM (TCAM). If more than one entry is matched, a priority encoder can be used to select the highest priority matching entry, where entries can have explicit priorities, or are ordered by position.

At first glance, CAMs seem to be the ideal solution to packet classification, as they meet the asymptotic lower bounds on the classification problem: a CAM-based solution to classification for N rules, with b bits examined uses $\Theta(N \cdot b + N \cdot \log N)$ gates ($\Theta(N \cdot b)$ for storing the rules, and $\Theta(N \cdot \log N)$ for the priority encoder) and has a delay of $\Theta(b + \log N)$ time ($\Theta(b)$ for the signal to propagate down the CAM word match line, and $\Theta(\log N)$ for the priority encoder).

However, the asymptotic notation hides constants, and in this case veil the true story. We will now argue that layout and circuit constraints make CAMs distinctly inferior. We do so by comparing a CAM-based solution to our solution for IP forwarding.

The BDD-based solution we described in Section 3 operates on a forwarding table with 57668 entries with a pipeline of 8 SRAMs, each 512 kbits in size.

A CAM-based scheme requires $57668 \cdot 32 = 1855936$ CAM cells. These are ternary CAM cells, which are of the order of $10\times$ the size of SRAM cells, so the CAM solution is approximately $5\times$ the BDD-based solution. (The ternary CAM cell is larger for several reasons: two obvious reasons are the need for two state holding elements, and the need for comparator logic.

Subtler reasons include the ability to use sense amplifiers in SRAM arrays, which allow for the SRAM transistors to be smaller sized; the sense amplifier area is amortized over a large number of SRAM cells [19].) Similarly, the access time to a 32 bit TCAM is approximately 10× that of SRAM.

However, there is a subtle but significantly greater problem with a CAM-based solution. Because of pipelining, the BDD-based solution can perform one forwarding decision per SRAM access time. A CAM solution can also be pipelined but at much higher cost: if we insert a pipeline flip-flop after *every* CAM cell, we need as many pipeline latches as there are CAM cells. Flip-flops burn a great deal of power, and now there is also the associated clock tree, which consumes power, and introduces problems with skew. The problem introduced by pipeline flip-flops can be reduced to some extent by having fewer pipe stages, but this increases delay. In contrast, in our SRAM pipeline, we need pipe flip-flops after each SRAM *stage*, and not each SRAM cell; thus flip-flops cost is negligible in our architecture.

5.2 Tree-like search structures

The packet classification is very similar to the point location problem. This has been studied for decades by computational geometers [1]. The best algorithms for this problem either have exponential space complexity and logarithmic time complexity, or linear time and space complexity (the exponential factor is dictated by the dimension of the space, which in the context of packet classification is the number of fields in the header). Conceptually, approaches either proceed one rule at a time (which involves multiple queries of the bits, but only linear space), or query the bits only once, but have exponential space.

Given the maturity of the computational geometry area, it is unlikely that there will be any breakthroughs (in the computational complexity sense) on the general form of this problem. Thus any improvements are heuristic, and justified on real-world rule-sets.

Given the time constraints imposed by link speeds, all recent schemes employ a lookup structure with reduced time complexity, and use heuristics to control space usage.

The approach taken in [22, 7] is, in essence, to build a *free BDD* [21] for the classification function. A free BDD differs from a BDD as defined in Section 2 in that the variable order along different paths may be different. Wegener [21] exhibits Boolean functions for which there is a polynomial sized

free BDD, but under all variable orderings the BDD is exponential sized. It is not clear how relevant this result is to functions that arise in the context of packet classification. Furthermore, there is no synthesis methodology, i.e., no description as to how to systematically build the free BDD. Finally, such a scheme is much more difficult to pipeline, as the order in which the bits are to be fed into the different stages is not fixed.

The approach taken by RFC [7] is, in essence, to build a *BDD tree* [13]; again, no synthesis methodology is presented. Although McMillan introduced BDD trees a decade ago, judging by citation information, very few researchers are aware of them. Essentially, instead of a total ordering on the variables, a binary tree is used, whose leaves are the variables, and internal nodes encode the possible cofactors with respect to the variables in the support of the node. Specifically, there is a matrix at each node, indexed by the cofactors with respect to the left subtree variables and the right subtree variables, and filled with the resulting cofactor. McMillan conjectured that for certain functions BDD trees were exponentially more succinct than BDDs could be. We implemented a BDD tree package to explore the benefits of tree BDDs. For the special case of a skewed binary tree, the tree BDD is isomorphic to a regular BDD, and so there is no benefit. For more balanced decompositions, the tree BDD was much larger than the BDD for functions from the ISCAS benchmark suite. The reason for this was that at a node even when the number of left and right cofactors is relatively small (say 1000 each), the matrix at the node becomes very large (1000×1000). This is true even when the number of distinct cofactors at the node is small. Hence, we concluded that BDD trees are not very useful for general functions.

BDDs and finite automata are intimately related; see [21], particularly Chapter 10. Our partitioned classifier in Section 4 essentially amounts to a finite automaton with bounded nondeterminism [8]; reading input bits out-of-order is related to the theory of 2-DFA [8]. Narayan *et al.* [15] proposed a straightforward algorithm based on enumerating co-factors for representing a logic function as a union of BDDs. Abstractly, this amounts to representing a regular language by the union of a finite number of DFAs which are pair-wise disjoint, i.e., using bounded nondeterminism. In our work we also represent a language as a union of a finite number of DFAs; however, since we focus on the class of functions corresponding to Layer-4 IP classifiers, we can use a great deal of domain knowledge to compute a good representation.

6 Conclusion

We described an architecture and associated synthesis methodology for Layer-4 packet classification. Based on published figures for modern SRAM performance and density, we argued that it should be feasible to build a chip that can be used to classify 500 million packets per second on a rule-set containing 10000 rules. This is an order of magnitude faster the fastest existing solution, and we achieved this performance by making use of parallelism and pipelining. Integral to our approach was the development of algorithms for optimally mapping rule-sets onto the architecture that exploited the fact that the classification function has a special form involving prefix matching.

We are also investigating generalizing our techniques to richer classification problems. For example, in this paper we have addressed “stateless” classification, i.e., the action taken for an incoming packet depends solely on the rule-set and the packet, and is not affected by packets which arrived previously. However, stateless classification is not powerful enough for certain applications, notably traffic-shaping and identifying denial-of-service attacks. Another limitation of this work is that we perform classification based solely on the Layer-4 header fields of the packet; for applications such as virus-detection and quality-of-service the classifier needs to look within the packet, and perform some form of regular expression search.

Appendix A: NP-completeness proof

The rule partitioning problem can be abstracted in terms of strings as follows. Given a set $R = \{r_i\}$ of ternary strings r_i , size of partition p and size of index set T , determine whether there exists a partition of R into sets $S_1, S_2 \dots S_p$ and an index set I_i consisting of integers between 1 and L for each set S_i such that, $|I_i| < T$ and for any pair of strings $r_j \in S_i, r_k \in S_i$ there is at least one index $l \in I_i$ such that $r_j[l] \neq r_k[l]$ and neither of them is an $*$.

Given an instance of this problem and solution it is easy to verify the correctness in polynomial time by looking at each pair of rules and each index bit to make sure that there is at least one index bit where the two rules differ. Thus the rule partitioning problem is in NP .

Partition into triangles problem for a graph $G(V, E)$ with $|V| = 3N$ is to determine whether a partition of V into $V_1, V_2 \dots, V_N$ exists, such that for any $V_i = \{u, v, w\}$, all three edge $\{u, v\}$, $\{u, w\}$, and $\{w, v\}$ are in E .

In this section we show that rule partitioning problem is at least as hard as partitioning into triangles, which is known to be NP-complete [4].

Given an instance of triangle problem on graph $G(V, E)$, such that $V = \{v_1, v_2, \dots, v_{3N}\}$ and $E = \{e_1, e_2, \dots, e_m\}$, construct a set of ternary strings $R = \{r_i\}$ such that there is one string r_i of length $|E|$ for each vertex v_i . For all $0 < i \leq m$, if $e_i = (v_j, v_k)$ and $j < k$ then set $r_j[i] = 0, r_k[i] = 1$, and for all other strings $r_o, r_o[i] = *$. Let $P = \{S_i\}$ be a partition of R . So for any pair of strings $r_j \in S_i, r_k \in S_i$, there must be at least one bit position l such that $r_j[l] \neq r_k[l], r_j[l] \neq *,$ and $r_k[l] \neq *$. Hence (j, k) is an edge in E . So each S_i corresponds to a clique in the graph $G(V, E)$. As we fixed number of index bits equal to 3 and number of partitions equal to N then any partition can at most contain 3 rules as more than 3 strings can not be differentiated by just 3 index bits (one index bit can only differentiate between a pair of strings) and it must be exactly 3 strings because there are only N partitions and $3N$ rules. Hence we can obtain a partition of R only if $G(V, E)$ can be partitioned into triangles. Similarly any partition of G into triangles will yield a partition of R . Hence any instance of graph partitioning to triangles can be reduced to rule partitioning.

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] R. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35:677–691, August 1986.
- [3] T. H. Cormen, C. E. Leiserson, and R. H. Rivest. *Introduction to Algorithms*. MIT Press, 1989.
- [4] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Co., 1979.
- [5] P. Gupta. *Algorithms for routing lookups and packet classification*. PhD thesis, stanford, 2000.
- [6] P. Gupta, S. Lin, and N. McKeown. Routing Lookups in Hardware at Memory Access Speeds. In *IEEE Infocom*, 1998.
- [7] P. Gupta and N. McKeown. Packet Classification using Hierarchical Intelligent Cuttings. In *Hot Interconnects*, Stanford University, CA, August 1999.
- [8] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
- [9] S. Keshav. *An Engineering Approach to Computer Networking*. Addison-Wesley, 1997.
- [10] S. Keshav and R. Sharma. Issues and Trends in Router Design. *IEEE Communication Magazine*, 1998.
- [11] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan-Kaufmann, 1991.
- [12] P. McGeer, K. McMillan, A. Saldanha, A. Sangiovanni-Vincentelli, and P. Scaglia. Fast Discrete Function Evaluation. In *International Conference on Computer-Aided Design*, November 1995.

- [13] K. McMillan. Hierarchical representations of discrete functions with applications to model checking. In *Computer Aided Verification*, July 1994.
- [14] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw Hill, 1994.
- [15] 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.
- [16] S. Northcutt and J. Novak. *Network Intrusion Detection: An Analyst's Handbook*. New Riders, 2000.
- [17] R. Panigrahy. Cisco Systems. Personal communication, 2002.
- [18] R. Perlman. *Interconnections*. Addison Wesley, 2000.
- [19] Jan Rabaey. *Digital Integrated Circuits: A Design Perspective*. Prentice Hall, 1996.
- [20] 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.
- [21] I. Wegener. *Branching Programs and Binary Decision Diagrams*. SIAM, 2000.
- [22] T. Y. C. Woo. A modular approach to packet classification: Algorithms and results. In *INFOCOM (3)*, pages 1213–1222, 2000.

Captions

Figure 1 SRAM implementation.

Figure 2 (a) BDD for the Boolean logic function $f = x_0 \cdot x_1 + \bar{x}_0 \cdot (\bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2)$. This corresponds to the forwarding table $\{(*, 0), (11*, 1), (001, 1), (010, *)\}$; addresses are 3 bits, and there are two output ports. (b) Representation of the BDD in memory.

Figure 3 Representation of a classification function by 2 BDDs. The first value at the terminals is the action; the second its priority.

Figure 4 Scaling of BDD size with number of prefixes.

Figure 5 Distribution of BDD nodes

Figure 6 Trade-off curve for number of MDD levels *vs.* minimum memory required to represent the MDD in memory banks

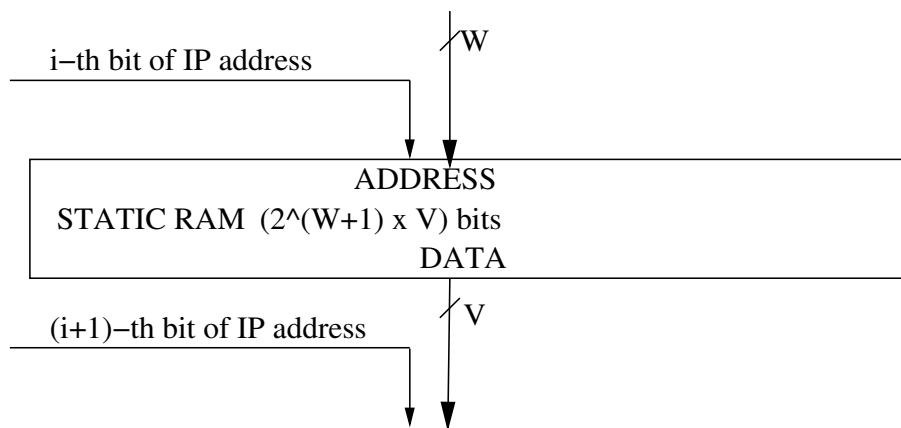
Figure 7 The power of partitioning.

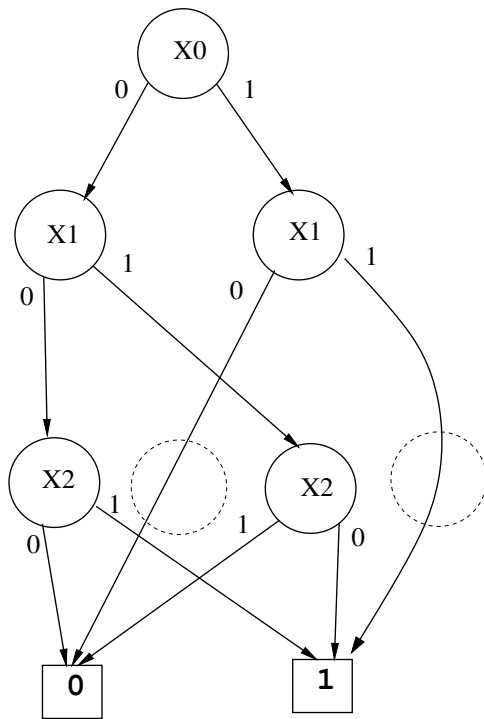
Figure 8 Architecture for evaluation of multiple BDDs. The individual subsets of the rule-set are evaluated using a monolithic BDD, exactly as in Section 3. The action taken is that of the highest-priority matching rule. Since the variable ordering may be different in different BDDs, we need to apply a permutation network to the inputs.

Figure 9 Scaling of BDD size, as measured in the total number of BDD nodes.

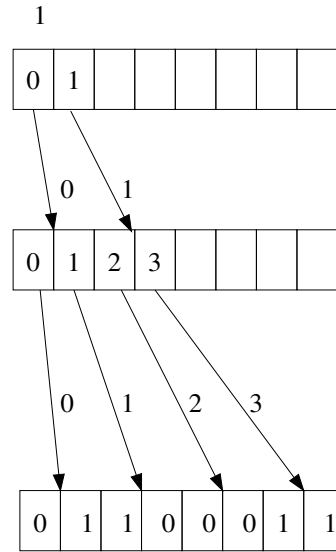
Figure 10 Scaling of size of memory banks, as measured by the largest width of any BDD.

Table 1 A rule-set on two fields.

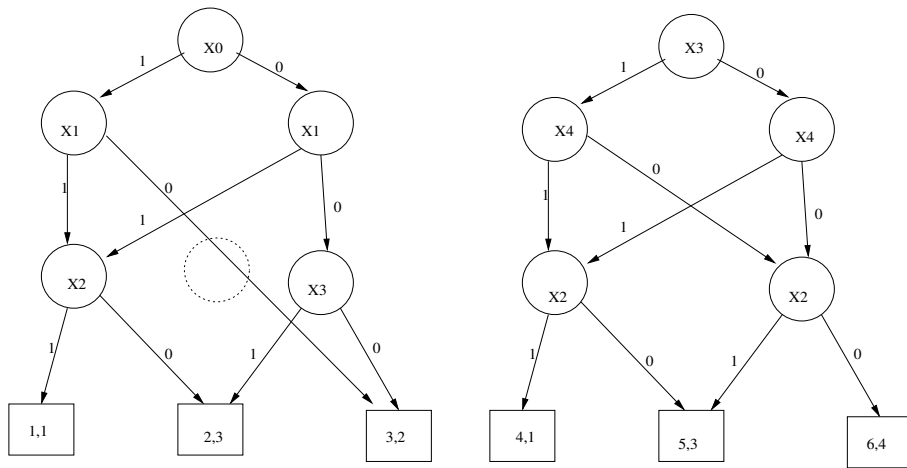




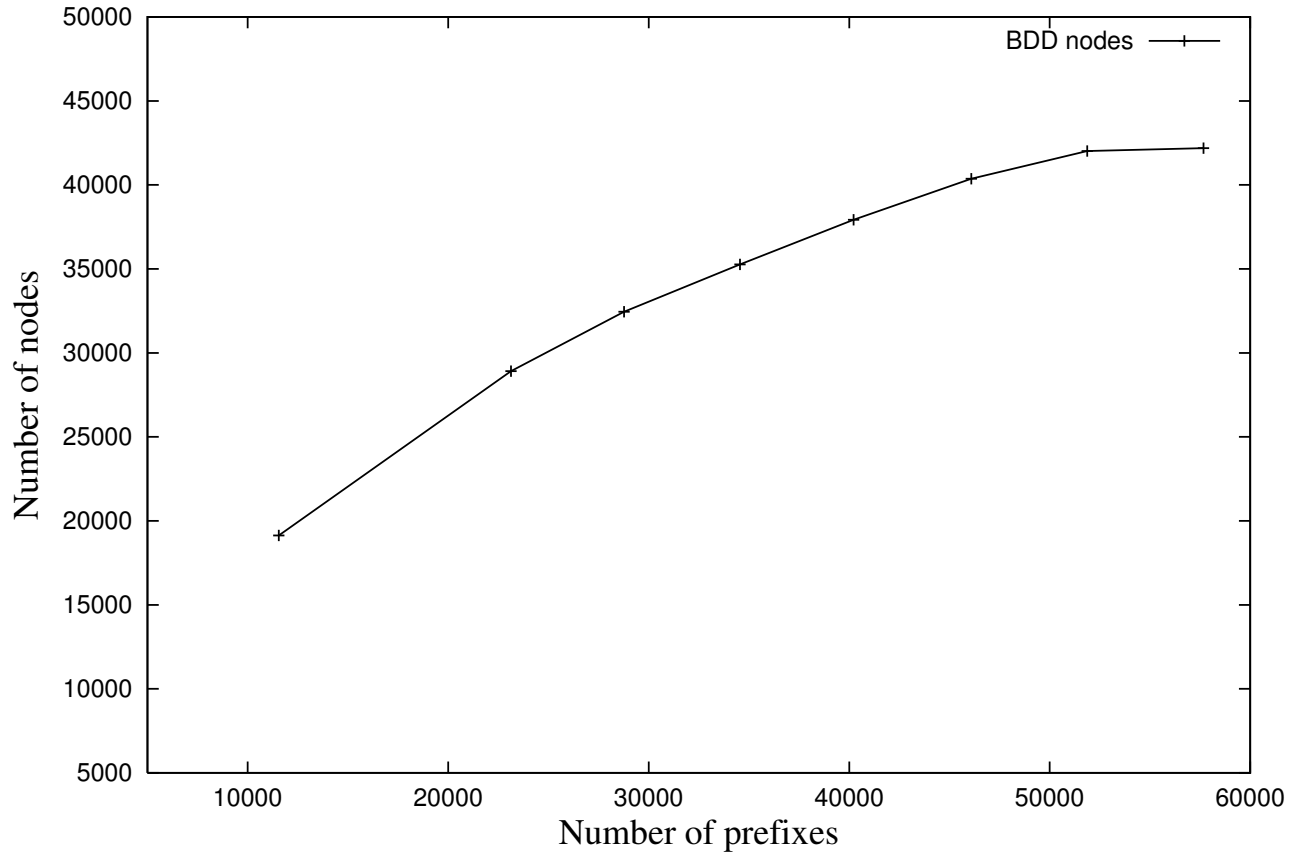
(a) BDD

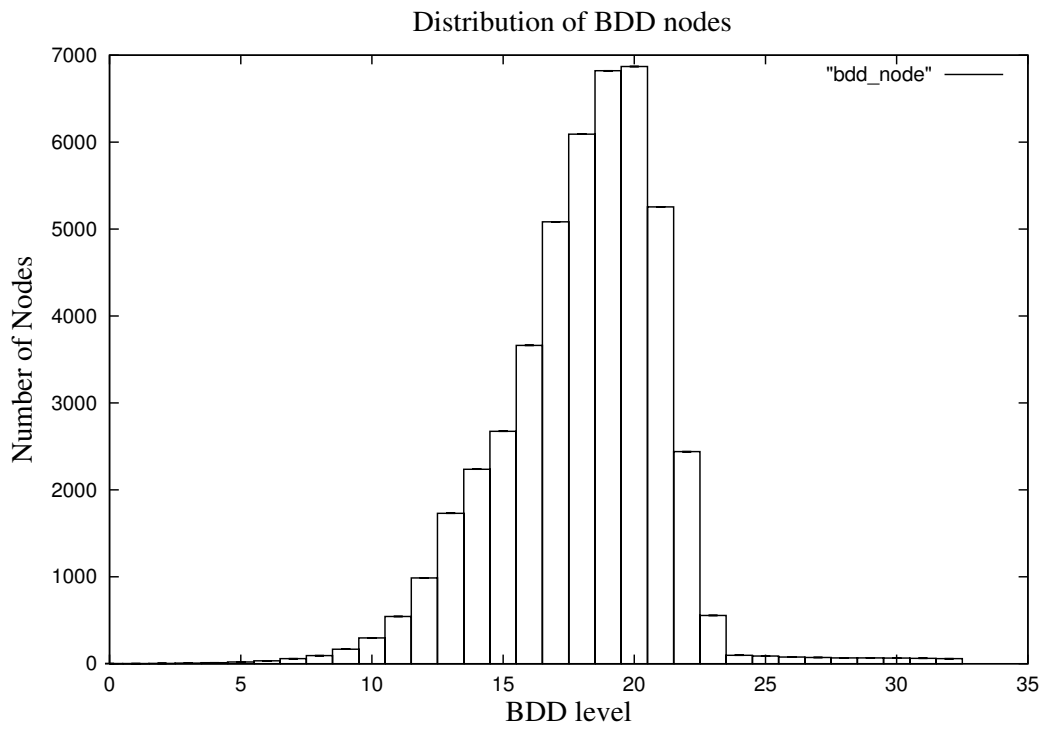


(b)

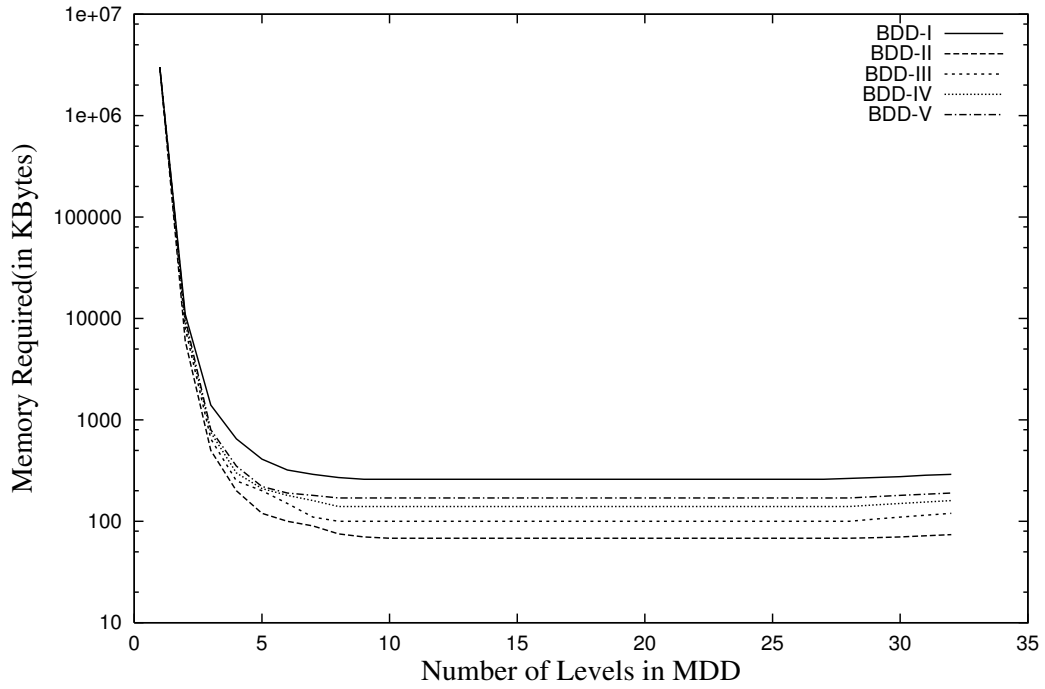


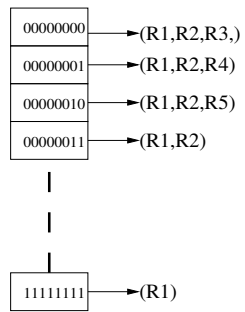
Number of nodes vs Number of prefixes



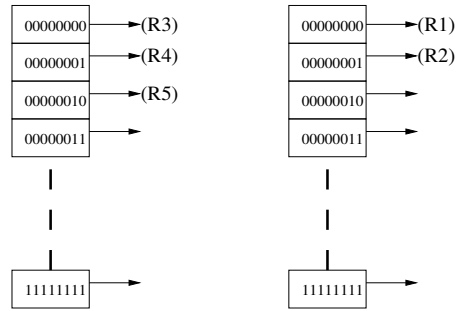


Memory Usage vs Number of Levels in MDD

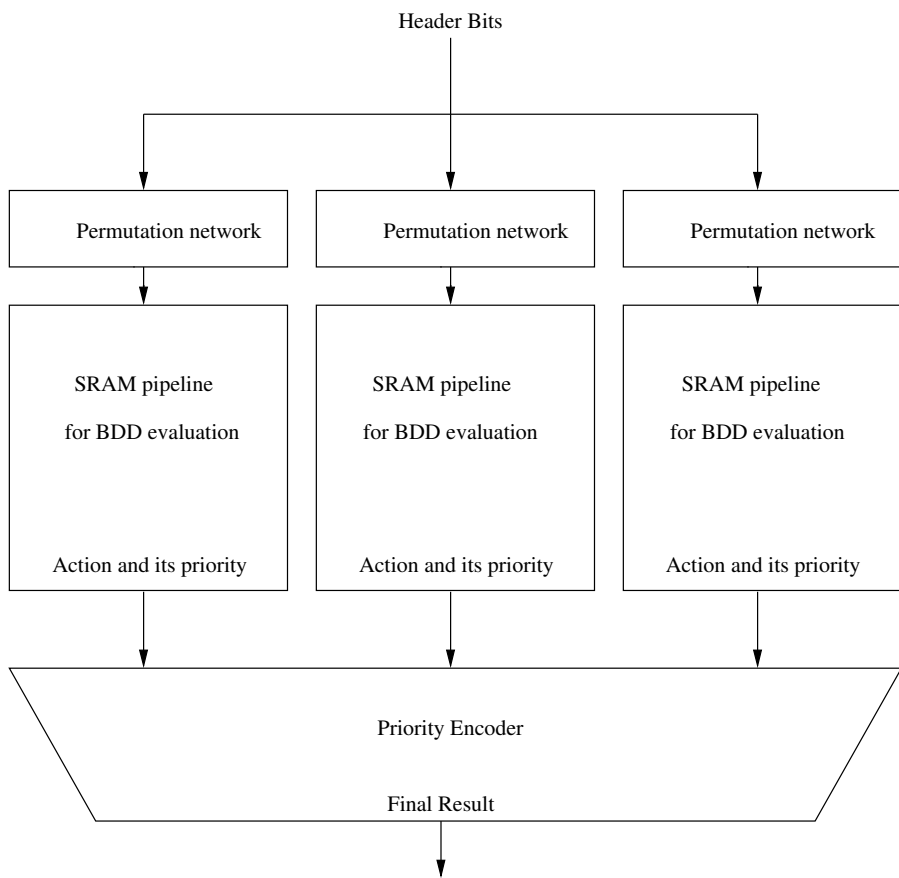


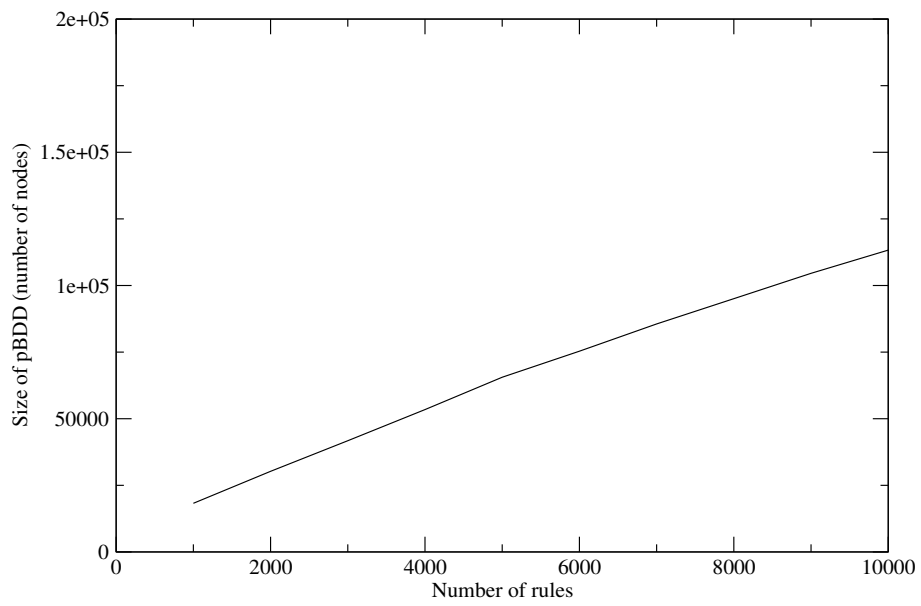


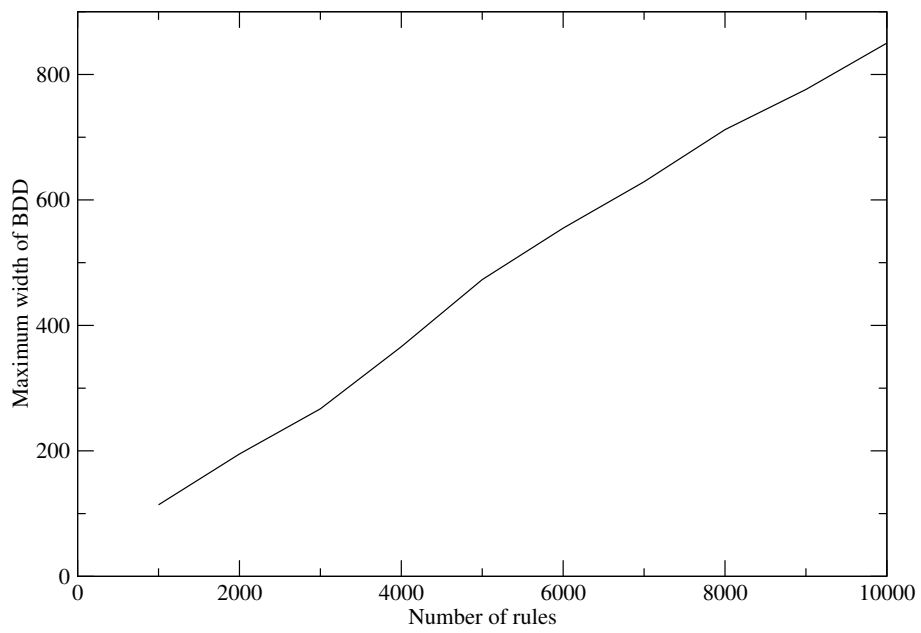
(a) Indexing on Source IP



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







Predicate	source IP	Destination IP
R1	*****	0000000
R2	00000**	0000001
R3	0000000	0000***
R4	0000001	110101**
R5	0000010	000*****

Figure 1: SRAM implementation.

Figure 2: (a) BDD for the Boolean logic function $f = x_0 \cdot x_1 + \bar{x}_0 \cdot (\bar{x}_1 \cdot x_2 + x_1 \cdot \bar{x}_2)$. This corresponds to the forwarding table $\{(*, 0), (11*, 1), (001, 1), (010, *)\}$; addresses are 3 bits, and there are two output ports. (b) Representation of the BDD in memory.

Figure 3: Representation of a classification function by 2 BDDs. The first value at the terminals is the action; the second its priority.

Figure 4: Scaling of BDD size with number of prefixes

Figure 5: Distribution of BDD nodes

Table 1: A rule-set on two fields.

Figure 6: Trade-off curve for number of MDD levels *vs.* minimum memory required to represent the MDD in memory banks

Figure 7: The power of partitioning.

Figure 8: Architecture for evaluation of multiple BDDs. The individual subsets of the rule-set are evaluated using a monolithic BDD, exactly as in Section 3. The action taken is that of the highest-priority matching rule. Since the variable ordering may be different in different BDDs, we need to apply a permutation network to the inputs.

Figure 9: Scaling of BDD size, as measured in the total number of BDD nodes.

Figure 10: Scaling of size of memory banks, as measured by the largest width of any BDD.