OC-3072 Packet Classification Using BDDs and Pipelined SRAMs

Amit Prakash Adnan Aziz
Department of Electrical and Computer Engineering
The University of Texas at Austin
prakash|adnan@ece.utexas.edu

Abstract

We present a solution to the problem of quickly classifying packets. Our approach is based on techniques from logic synthesis. Specifically, we express the classification rules as Boolean logic equations, build Binary Decision Diagrams for these equations, and then map the BDDs to a logic network consisting of a pipeline of static RAM banks. We illustrate our approach by applying it to the longest prefix matching for IP forwarding, and present evidence that our scheme can perform a billion matches per second on a CAIDA backbone forwarding table containing 60,000 prefixes. We show how our approach generalizes to classifying packets on multiple fields.

1 Introduction

Until relatively recently, routers were little more than general purpose computers connected to specialized hardware for transmitting and receiving packets over links. This was because link bandwidth was low enough that general purpose processors could implement all the functionality needed for routing.

The advent of high-speed optical link technology has led to a reversal to this situation — today routers and not links are the bottleneck in moving information around the Internet. One approach to make routers faster is to implement performance-critical aspects of routing in custom hardware.

One of the basic operations that a router has to perform is to take an incoming packet and determine which output link to put it on. The "forwarding table" contains the information needed to make this decision. Conceptually, this table consists of a set of *(bitPrefix, outputPort)* pairs. The 32 bit IP destination address d of an input packet p is compared with the prefixes in the set and the packet p is forwarded to the output port that corresponds to the longest prefix that matches d. Routers participate in elaborate protocols to compute forwarding tables which result in paths that are in some sense optimum [8, Chapter 11].

We focus on one of the most performance critical computations performed by a router, namely packet classification, a special case of which is the longest prefix matching problem previously described. In its more general form, the problem consists of looking at multiple fields in the packet header, and determining what actions to perform on the packet. In addition to making forwarding decisions, packet classification has applications to implementing class-of-service, building firewalls, gathering statistics, enforcing service-level-agreements, etc.

To keep our exposition simple, we will first illustrate our approach on the longest prefix matching problem. We will describe how our approach generalizes to classification on multiple fields to the general problem at the end of the paper.

1.1 Prior work

The longest prefix matching problem has received widespread attention. Approaches can be grouped into two classes: software-centric, e.g., [1, 9, 12, 11] and hardware-based, e.g., [5, 6].

The state-of-the-art in software-based solutions is embodied by the binary search on hash tables algorithm [12]. Since the data structures are large, its best case performance is bounded by the latency of dynamic RAM, which is approximately 50 nanoseconds. In practice, the approach is reported to achieve approximately 2 million matches per second, which is too slow for today's high speed optical links. Furthermore, incremental updates are extremely complex; given that backbone routers change their forwarding table based on BGP updates every 30 seconds, this is a major limitation. Finally, the approach involves relatively complex operations (e.g., computing hash codes) and is not "regular" enough to be easily mapped into a direct hardware implementation.

A diverse set of hardware-based solutions have been offered to the longest prefix matching problem. Gupta *et al.* [5] describe a scheme which expands all the prefixes up to 24 bits in length, and stores that portion of the for-

warding table in DRAM. Since the vast majority of prefixes are no more than 24 bits long, for these prefixes they can make a forwarding decision by simply doing a DRAM access, thereby achieving up to 20×10^6 matches per second for 50 nanosecond DRAM. However, their approach suffers from several limitations: it employs a large amount of (power hungry) DRAM (9–33 Mbytes are reported), lookup times depend on the prefix length distribution, and updates are complex. The approach does not scale — it cannot be used for IP version 6, or for level 4 packet classification. Another hardware oriented approach is the use of Content Addressable Memories (CAMs) [6]. A CAM is a fully associative memory, i.e., it can perform an exact match in a single clock cycle by doing multiple comparisons in parallel. Longest prefix matching can be performed using Ternary CAMs with some priority decode logic. CAMs can be used to compute up to 50 million matches per second. However, updating the CAM is difficult, since entries need to be ordered by length. Furthermore, CAMs, being latch based, burn a great deal of power.

1.2 Problem relevance

It is worth stressing that the longest prefix matching problem is still important today. There are two arguments that have been made that it is irrelevant: (1) since packets are written in slow DRAM memory, this is more of a bottleneck to performance, and (2) deployment of multiprotocol label switching (MPLS) does away with the need for doing longest prefix matching.

We argue against the first point by noting that a fast matcher can be used to perform the forwarding decisions for multiple input ports. Furthermore, the general packet classification problem is more complex, and needs correspondingly more computation.

Similarly, the introduction of MPLS does not completely do away with the packet classification problem. It is not clear whether MPLS will really be widely deployed, as it requires a complex label management scheme in a distributed environment. MPLS labels will still need to be computed at entry points into MPLS networks. Furthermore, more general classification (level 4 and above), cannot be achieved by simple MPLS labels because MPLS labels are simply too "coarse."

2 Background — logic synthesis & BDDs

Logic synthesis [10] is the term given to the process of realizing an optimized gate-level implementation of a logical specification. It is common to express the specification using Boolean logic equations. The task of the synthesis tool is to compute an optimized netlist of logic gates, drawn from a target standard-cell library, which implements these logic equations.

Truly optimum synthesis is extremely difficult to achieve, due to the fact that the underlying decision problems are invariably NP-hard. As such, general purpose synthesis tools make extensive use of heuristics. In our work, we will develop a logic synthesis procedure that maps the forwarding table expressed using Boolean logic equations to a special reprogrammable architecture that is suitable for implementing classification rules.

Given that synthesis tools operate with Boolean-valued functions of Boolean-valued variables, it is imperative to use a data structure that can compactly represent and manipulate a large class of useful Boolean functions. The data structure of choice for representing Boolean functions is the Reduced Ordered Binary Decision Diagram [2].

Binary Decision Diagrams have their roots in the decomposition given by the Shannon expansion theorem, i.e., the result that $f = x \cdot f_x + x' \cdot f_{x'}$. Recursively applying this decomposition leads to a tree structured representation. A reduced ordered binary decision diagram (henceforth BDD) for the function, 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, nodes with equal children are removed, and isomorphic subtrees are merged. An example of a BDD is given in Figure 1(a). The two children of a BDD node are referred to as the 0-branch and 1-branch; these correspond to the function computed when the node variable is set to 0 or 1, respectively.

3 Our classifier

Let us consider a router which has 2^k output ports. Given a forwarding table, our goal in the longest prefix matching problem is to find an optimized implementation of the function $\mathcal{F}_{\text{PM}}$ which takes as an argument a 32 bit address (i.e., has domain $\{0,1\}^{32}$) and returns a k-bit output port identifier, i.e., has range $\{0,1\}^k$.

Given a forwarding table, it is straightforward to write Boolean logic equations specifying $\mathcal{F}_{\text{PM}}$. In principle, we can run logic synthesis on these equations to obtain an efficient hardware implementation of $\mathcal{F}_{\text{PM}}$. Performing longest prefix matching is then just a matter of floating the destination IP address as an input to this hardware — the output port identifier is the output of the synthesized circuit.

This approach differs from previous hardware approaches in that the forwarding table is encoded in the circuit itself, instead of being fed as an input to a logic circuit. However this also means that hardware has to be reprogrammable as the table may change. Thus, our logic synthesis algorithm needs to target reprogrammable hardware.

For reprogrammable hardware, Field Programmable

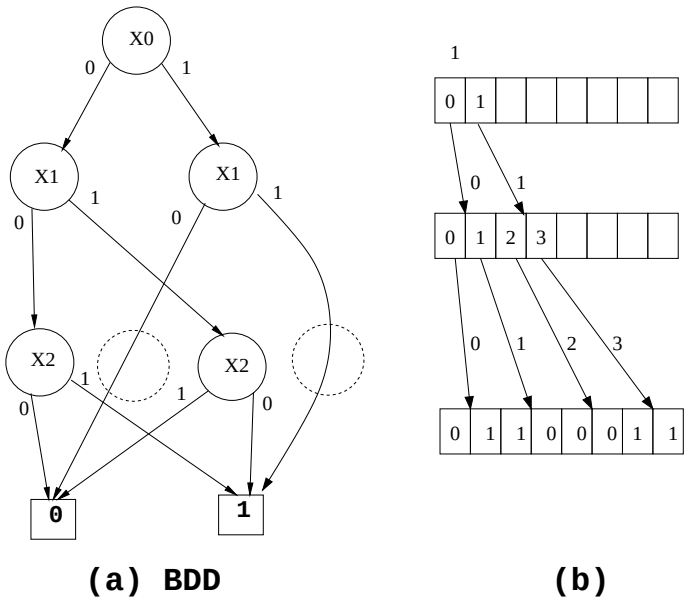

Figure 1. (a) BDD for the Boolean logic function $f = x_0 \cdot x_1 + \bar{x_0} \cdot (\bar{x_1} \cdot x_2 + x_1 \cdot \bar{x_2})$. (b) Representation of the BDD in memory.

Gate Arrays (FPGAs) may seem to be the obvious choice, but we found them to be ill-suited for this application. We attempted to map logic equations corresponding to a backbone 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 $\mathcal{F}_{\text{PM}}$, 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 $\mathcal{F}_{\text{PM}}$), 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.

In summary, using generic logic synthesis on the logic equations corresponding to the forwarding table is not possible because a hard combinatorial problem has to be solved, and traditional reprogrammable architectures are not suited to implementing the equations. In the next section, we describe our approach, which overcomes both these problems.

3.1 Our architecture: cascaded SRAMs

To illustrate our approach, we first consider the case where there are exactly two output ports, i.e., $\mathcal{F}_{\text{PM}} : \{0,1\}^{32} \mapsto \{0,1\}$.

Our approach consists of building the BDD for $\mathcal{F}_{\text{PM}}$ and then mapping it to a pipeline of 32 SRAMs, numbered from 0 to 31, as in Figure 2. 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 1(b).

We have shown that the size of the BDD representation of a forwarding table is bounded by the size of the forwarding table. Due to space restrictions, we omit a formal proof of this result. (In fact the BDD will actually be significantly smaller than the forwarding table because of node sharing.)

As an example, consider the forwarding table: $\{(0*,0),(1*,1),(01*,1),(011,0),(10*,0),(011,0)\}$. The BDD corresponding to this forwarding table mapping can be translated to the BDD drawn with solid lines in Figure 1(a). When we want to translate this BDD into the cascaded memory architecture, we will have to add extra nodes on edges which skip levels. The dashed circles represent the new nodes. Figure 1(b) shows how these nodes can be arranged in different SRAM banks.

A similar architecture can also be used to perform a trie walk in hardware. However for our application, the BDD representation is considerably smaller than the corresponding trie because of node sharing. This is especially true at the lower levels, where the width of the trie is much greater than that of the BDD for same forwarding table. The small size of the BDD lets us use very fast SRAMs, which would otherwise be infeasible. Furthermore, since a trie can skip multiple levels, it is impossible to pipeline

We now provide a more detailed description of the architecture. Let us assume that maximum number of BDD nodes at any level is bounded by 2^c nodes where c is some constant. Let each SRAM be capable of storing 2^{c+1} words, each c bits wide. The least significant address line of the i-th SRAM is driven by the i-th bit of input IP address. The remaining address lines of the $i+1$-th line is driven by the data-out lines of the i-th SRAM. Again, assume that each BDD node has its 0-child and 1-child in the next level. (Later we will show we can get rid of this restriction with a little overhead.) We store the nodes at level i in an address in SRAM i (each node is two words, a pointer to the 0-child and a pointer to the 1-child). Since the number of nodes is limited by 2^c we can do this. As there will be only one top node in the BDD we can let the $[c:1]$ address bits of SRAM 0 remain fixed. The matcher operates as follows: we float the IP address as the input. The first bank generates the

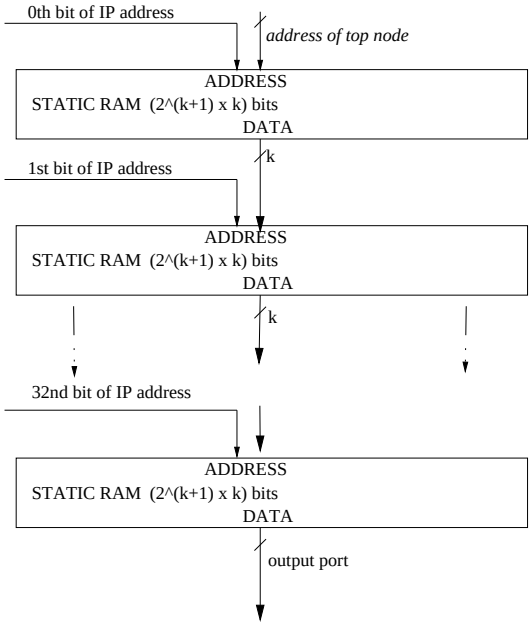

Figure 2. SRAM implementation.

higher address bits of the correct node in the second level. These bits combined with the second bit of IP Address go to the second SRAM and we get the required address bits of node in the third bank. This way when we get all the way to the last bank we get the correct port out. Since all the SRAM blocks have same delay we can easily pipeline this architecture so that data coming out of each stage is latched for the next clock cycle.

In the next section, we will show experimentally that if we allocate 16K BDD nodes for each level, we can accommodate 60K prefixes with room to spare. Hence it suffices for each SRAM to be $32K \times 14$ bits in size. Thus a 64 Kbyte SRAM will easily serve our purpose. Modern L2 caches are of this size with a cycle time as low as 0.66 nanoseconds per access. (For example, the Pentium-IV operates at 1.5 Ghz, and has a single cycle access time to the L2 cache [7].) So if each pipeline stage just does one memory access, we can do one lookup every 1 ns, i.e., a billion lookups per second. (We are budgeting of 0.33 ns for interconnect delay; given that the communication is very local, this should be more than adequate.) Assuming 160 bit IP packets, we can serve a line operating at $160 \times 10^9 = 160$ Gbps, i.e., an OC-3072 link.

The above discussion has assumed that there were two output ports, with ids 0 and 1. We can generalize it to k-bit output port ids by building k BDDs, one for each bit of the output port id. The disadvantage with this approach is it requires $(k \cdot 32)$ SRAMs.

However there is a simple trick that allows us to get by with 32 SRAMs even for k-bit output port ids. Instead of having just two nodes, zero and one, at the last level of the BDD, we will have 2^k nodes, one for each output port. (This data structure is referred to as a multi-valued BDD (mvBDD) [2].) We can easily construct this data structure by building the BDD of the function $\mathcal{G}_{\text{PM}} : (IP\_address, port) \mapsto \{0, 1\}$ where $\mathcal{G}_{\text{PM}}$*(ip,port)* = 1 iff the IP address *ip* matches the port number *port*. While building this BDD, we force the variables corresponding to the port id bits to be the lowest variables of the BDD. Then if we cut the BDD at the level where port-id bits start, we get the corresponding mvBDD.

Reducing the number of levels

We note that if we are allocating 16K nodes for each level, we do not need the first 14 SRAMs — we can use the first 14 bits of IP address to directly index to the corresponding nodes at level 14. For a given 14 bit prefix of IP address we can walk down the original BDD and get the corresponding BDD at 14-th level. So we just write that node at the address formed by the 14 bit prefix. (Some of the nodes may have multiple copies in the same level but that is acceptable because number of nodes cannot exceed the total size of memory.)

Another observation, in the spirit of Gupta *et al.* [5], is that almost prefixes are less than or equal to 24 bits. So in most cases we can get the result after the 24-th level. After that we can have one bit to indicate whether we are done. In case we are not done, we can use the remaining 8 bits and 7 bits from previous pointer (if we provision for at most 128 prefixes longer than 24 bits). Thus we need only 11 SRAM banks, as opposed to 18. This significantly reduces the area requirements, power dissipation and latency of the system.

Removing restrictions on BDDs

We previously imposed a restriction on BDDs that each node must have its 0-branch and 1-branch nodes in the next level. However this is not the case for standard BDDs, where edges can skip level. There are two solutions to this problem. The trivial solution is to insert extra nodes on edges which skip levels, as shown in Figure 1. An alternative approach is to use 2 bits with each pointer to tell how many nodes we can skip. With a little more hardware we can let each pipeline stage can check if the level skip bits are zero or not. If they represent zero, then it function as usual otherwise it just subtract the number of levels to skip and pass the same address. With 2 bits we can skip maximum up to 4 levels which will considerably reduce the number of replicated nodes.

Updates

Performing updates in our system is straightforward. When a prefix is changed, the number of nodes in the original BDD that change is always less than equal to the number of levels. Since we have just 11 levels in our modified decision diagram, in most cases we will have less than 12 nodes changing. Furthermore these words will be going to different banks so they can be written simultaneously. However in the modified decision diagram since we start directly at level 14, a single node can map to many addresses so a change in a class A prefix (8 bits) will lead to $2^{14-8} = 64$ entries of two words each which is not excessive, and such changes will be infrequent as there are very few class Addresses. Also because we expand all the bits from 25 to 32 in the last level, a change in 25 bit prefix can lead to $2^7 = 128$ words. Prefixes longer than 25 bits are also very infrequent. If we assume these problematic updates occur less than 10% of the time then average number of words to be written on the memory banks will be less than 35.

If we assume 1000 BGP updates come every 30 seconds, in the worst case we need to write $(128 + 10) * 1000 = 138K$ words and in average case $35K$ words. This is very little data over a 30 second time period — it can even be downloaded via a serial link. Inside the chip all the SRAMs can be written in parallel, so in the worst case one of the banks will write 128 words for an update, meaning 128K writes. If the write latency of SRAMS is 10 ns then this will make the classifier unavailable for $128K \times 10$ ns = 1.28 ms every 30 seconds. This reduced the throughput of the classifier by only $.00128/30 = 0.04\%$.

3.2 Hardware implementation

One of the strengths of our architecture is that it is very regular and consequently easy to implement. All we need is a number of SRAM banks placed one after another. SRAMs are standard components; vendors such as LSI Logic sell memory generators that create layout and timing models for parametrized SRAMs. The control logic to perform writes and multiplex addresses is minimal. For routing tables not needing more than 16K nodes at any level we can do with 11 SRAMs each 64 Kbytes in size. A 128 Kbyte SRAM in 0.18 micron CMOS technology is about 2500×3000 square microns in area, and dissipates 0.25 Watts, so if we have 6 such banks, the die size would be 7.5mm $\times$ 6mm and the power requirement would be around 1.5W (these numbers were obtained from a memory designer from a large semiconductor vendor).

Note that the total delay through the pipeline is 11 times the delay of each stage, i.e., 11 ns. This requires buffering 1760 additional bits for an OC-3072 line, and corresponds to a propagation delay of 2.2 meters through copper wire.

Bit	No. of nodes	Bit	No. of nodes
14	2164	20	6803
15	2678	21	5207
16	3675	22	2745
17	5096	23	1001
18	6093	24	335
19	6745		

Table 1. Number of nodes at different levels.

$\mid P \mid$	$\mid BDD \mid$	$\mid BDD[14:24] \mid$	$\mid M \mid$	*time*
11546	23308	18512	19128	7.2
23132	34065	28042	28918	9.3
28769	37847	31444	32445	11.4
34550	40855	34211	35264	13.3
40222	43777	36763	37926	15.2
46092	46370	39119	40363	17.0
51867	48189	40726	42014	18.7
57668	45063	40995	42188	20.6

Table 2. Scaling of system with number of prefixes. Here $\mid P \mid$ denote the number of prefixes, $\mid BDD \mid$ denotes the number of BDD nodes, $\mid BDD[14:24] \mid$ denotes the number of BDD nodes between level 14 and 24, $\mid M \mid$ is the total number of nodes in memory, *time* is CPU time in seconds to build the BDD on a PIII 500MHz/256Mbytes running Redhat Linux 6.2.

3.3 Experiments

We evaluated our approach with a forwarding table for MAE-WEST obtained from CAIDA [3]. The table has 57668 prefixes and 62 output ports. The table 1 shows the number of nodes we need in different SRAM banks to map this routing table. The maximum number of nodes needed is 6803 at level 20. So routing tables containing 60 K prefixes should easily fit in the SRAMs by provisioning memory for 16K nodes in each SRAM.

We can get some idea of how the total number of BDD nodes scales with the number of prefixes by considering subsets of the forwarding table selected randomly with similar distribution of prefix lengths. Table 2 shows how the number of BDD nodes scales with the number of prefixes. The second column shows number of nodes in the exact mvBDD. The third column is the number of nodes between level 14 and 24. The fourth column is the actual number of nodes in memory after accounting for extra nodes to needed to skip levels. From Table 2 it is clear that the number of nodes grows sublinearly with the number of prefixes.

4 Packet classification on multiple fields

We now turn to the general packet classification problem, which we illustrate by considering level 4 packet classification. Level 4 packet classification requires examining multiple fields in the packet header, and is significantly more complex than the longest prefix matching problem. In a common formulation [4, page 21], a level 4 classification rule consists of an ordered pair, with the first element being a predicate, and the second element being an action. The predicate is a conjunction of conditions on the source IP address, destination IP address, source port, destination port and protocol fields. The conditions themselves could be ranges (e.g., source port > 1023), equality (destination port = 80) or prefixes (source IP = $101*$). The action could be a 4-bit quantity denoting class-of-service, or even a single bit denoting whether to block the packet. In the common formulation, the set of rules is assumed to be totally ordered, and if for a given packet, two rules' predicates are true, then the action taken is that of the higher ranked rule.

The direct approach of building a BDD for the classification function computing the action as a function of the header, and then mapping the BDD to a pipeline of memory banks does not work well in case of generalized classification. The BDD size for each individual predicate will be small. However the fact that the rules may be ordered arbitrarily can result in enormous BDDs. For example, we generated 1000 random classification rules and found the resulting BDD to have more than a million nodes.

We can avoid this BDD blowup by slightly modifying the semantics of classification. Specifically, we allow the user to assign a priority level to each rule from a small, predefined set of priorities, e.g., $\{0, 1, \ldots, 7\}$. Then if there are multiple rules that match a packet, the highest priority rule will be applied. If more than one rule with highest priority is applicable to a packet, then the classifier is free to select any one of them to apply to the packet. We will shortly describe how to use this flexibility to order predicates of a given priority in such a way that the BDD for the classification function is linear in the number of rules of that priority. We believe that these restrictions will not be very significant for most real-world applications.

Since there is a known number of priority classes, we can build a pipeline of SRAMs computing the action for the set of rules at a priority level exactly as before. The classifier then consists of pipelines for each priority operating in parallel and a priority encoder, which then selects the highest priority action to apply.

We now provide details on how we order predicates within a priority level. First of all, the only conditions we consider are prefixes on individual fields. Both equality and range checks can be expressed as prefixes with at worst as many prefixes as number of bits in the field. (If there are ranges on multiple fields in a predicate and all the ranges need splitting, there can be a multiplicative effect. However in practice arbitrary ranges are used for port numbers only.) We define a prefix p_1 to be less than p_2 (denoted by $p_1 \prec_\pi p_2$) if and only if p_2 is a proper prefix of p_1. We say p_1 and p_2 are incomparable if $p_1 \not\prec_\pi p_2$ and $p_2 \not\prec_\pi p_1$. We totally order the individual fields, e.g., *source IP* $>$ *destination IP* $>$ *source port* $>$ *destination port* $>$ *protocol*. Given two predicates $A = (a_1, a_2, a_3, a_4, a_5)$ and $B = (b_1, b_2, b_3, b_4, b_5)$, we define $A \prec B$ if and only if $a_i \prec_\pi b_i$ where i is the smallest index such that a_i and b_i are comparable. Now if we make sure that rules are prioritized according to the order $\prec$ on the predicates, we can prove that the size of the Shannon tree of resulting classification function will be linear in number of rules. A BDD is a reduction of the Shannon tree, so it will also be linearly bounded in number of rules (and possibly much smaller because of sharing).

References

[1] A. Brodnik, S. Carlsson, M. Degermark, and S. Pink. Small forwarding tables for fast routing lookups. In *ACM SIGCOMM*, 1997.

[2] R. Bryant. Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification. In *Proceedings International Conference on Computer-Aided Design*, November 1995.

[3] CAIDA. `www.caida.org`.

[4] P. Gupta. *Algorithms for routing lookups and packet classification*. PhD thesis, CS Department, Stanford University, 2000.

[5] P. Gupta, S. Lin, and N. McKeown. Routing Lookups in Hardware at Memory Access Speeds. In *Proceedings IEEE Infocom*, 1998.

[6] Lara Networks Inc. `www.laranetworks.com`.

[7] Intel. `developer.intel.com/design/Pentium4/prodbref/`.

[8] S. Keshav. *An Engineering Approach to Computer Networking*. Addison-Wesley, 1997.

[9] B. Lampson, V. Srinivasan, and G. Varghese. IP Lookups using Multiway and Multicolumn Search. In *Proceedings IEEE Infocom*, 1998.

[10] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw Hill, 1994.

[11] V. Srinivasan and G. Varghese. Fast IP Lookups Using Controlled Prefix Expansion. In *ACM SIGMETRICS*, 1998.

[12] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable High-Speed IP Routing Lookups. In *ACM SIGCOMM*, 1997.