Leaping multiple headers in a single bound: wire-speed parsing using the Kangaroo system

Christos Kozanitis
University of California San Diego
Email: ckozanit@cs.ucsd.edu

John Huber
Cisco Inc
San Diego, CA

Sushil Singh
Cisco Inc
San Jose, CA

George Varghese
University of California San Diego

***Abstract*—More fundamental than IP lookups and packet classification in routers is the extraction of fields such as IP Dest and TCP Ports that determine packet forwarding. While parsing of packet fields used to be easy, new shim layers (e.g., MPLS, 802.1Q, MAC-in-MAC) of possibly *variable* length have greatly increased the worst-case path in the parse tree. The problem is exacerbated by the need to accommodate new packet headers and to extract other higher layer fields. Programmable routers for projects such as GENI will need such flexible parsers. In this paper, we describe the design and implementation of the Kangaroo system, a flexible packet parser that can run at 40 Gbps even for worst-case packet headers. Because conventional solutions that traverse the parse tree one protocol at a time are too slow, Kangaroo uses *lookahead* to parse several protocol headers in one step using a new architecture in which a CAM directs the next set of bytes to be extracted. The challenge is to keep the number of CAM entries from growing exponentially with the amount of lookahead. We deal with this challenge using a non-uniform traversal of the parse tree, and an offline dynamic programming algorithm that calculates the optimal walk. Our experiments on a NetFPGA prototype show a speedup of 2 compared to an architecture with a lookahead of 1. The architecture can be implemented as a parsing block in a standard 400 MHz ASIC at 40 Gbps using less than 1% of chip area.**

I. Introduction

This paper introduces packet parsing as an important new packet forwarding bottleneck in high speed routers, and describes algorithms to combat this bottleneck by extracting and processing multiple protocol headers in a single step. The classical bottlenecks in a high speed router are IP lookups, packet classification, switching QoS processing. However, each of these processing steps is based on fields in the packet being processed. For instance, IP lookups depend on the IP Destination Address, packet classification is often based on the TCP port numbers, and both switch and QoS processing often requires access to the TOS bits. If we refer to *parsing* as the extraction of key packet fields such as IP Destination and TCP Ports, then it is obvious that parsing is even more fundamental than lookups or switching.

At first glance, parsing does not seem to be a bottleneck. If one envisions the simplest sequence of packet headers (Figure 1) where Ethernet is followed by a fixed length IPv4 header and then TCP, then field extraction is hardly an issue. Extracting the Ethertype field (field A in Figure 1 at an offset of 96 bits) and checking for EtherType = IPv4 allows extraction of the IP header starting at byte 14. The IP Destination address is easily extracted at an offset of 16 bytes from the start of the IP header. Further, extracting the IP length (field B in Figure 1) together with a check of the IP protocol field (field C) provides the start offset of the TCP header field, from which the TCP header can be extracted. Thus in 3 or 4 processing steps, some of which can be easily parallelized, all the key fields in the TCP and IP headers can be extracted.

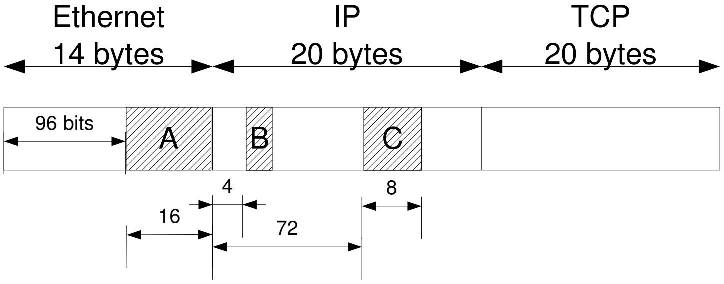

Fig. 1. A simple sequence of packet headers. Whenever unspecified, values of offsets are in bits.

Why Parsing is a Bottleneck today: However, parsing has become a problem because of three fairly recent developments. The first and biggest issue is the presence of a large number of intermediate headers used to add tags to packets (e.g., MPLS) or to create tunnels (e.g., GRE). These include a number of recent shim headers such as 802.1Q, several layers of MPLS including EOMPLS, and IP-in IP encapsulation.

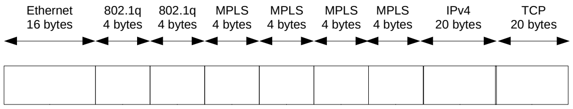

Fig. 2. A more complex but realistic sequence of packet headers

If we draw a simple protocol tree picture with each protocol as a node (Figure 3) then the sequence in Figure 1 corresponds to the leftmost path in the tree of length 3, while the sequence in Figure 2 corresponds to the rightmost path of length 9. If we intuitively represent the complexity of a parsing path by the length of the corresponding path in the protocol tree, then it is apparent that the packet in Figure 2 is 3 times harder to parse than the packet of Figure 1.

Complex packet headers such as depicted in Figure 2 are worth optimizing for because they are commonly used in practice. Most commercial routers are designed to handle

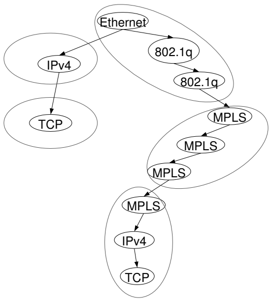

Fig. 3. A protocol tree representing the packet headers in Figure 1 and Figure 2. The ovals represent potential lookahead processing opportunities.

various combinations of these headers. First, 802.1Q shim headers are universal because of so-called VLAN tagging that allows customers [10] to multiplex several logical networks for each department on a single corporate network without fear of leaking or snooping. Many ISPs have two layers of 802.1Q (double tagging) because of the need to mix internal VLANs with customer VLANs.

Next, MPLS has become a de facto standard for traffic engineering by creating a so-called tunnel which allows the tunneled traffic to be mapped onto specific queues within a sequence of routers. MPLS allows tunnels to be created within tunnels so that traffic from various edge routers can be routed on first level tunnels, that are then aggregated on to second-level tunnels between regions, and so-on. This form of hierarchical tunneling is implemented by stacking sequences of MPLS headers, and 4 levels are commonly supported by most routers. Multiple levels are especially needed to support VPNs, which are routinely deployed by ISPs today. [16] shows that one can considerably reduce the overall number of labels used in the network by increasing the stack depth, and that the optimum stack depth is around 4.

Thus every commercial vendor optimizes for processing headers such as Figure 2 at wire-speed. Cisco [4] supports 6 levels of MPLS and at least two levels of VLAN tags at 40 Gbps. Juniper [5] supports 5 MPLS headers on T-series, in addition to two levels of VLAN tags. However, neither support *programmable parsing* at 40 Gbps.

IPSec also allows the use of security headers to create encrypted tunnels. Finally, Mobile IP and other protocols often use GRE encapsulation to add further information to packets. In sum, each shim header adds some useful information such as VLAN tags, traffic engineering tunnel tags, and security and mobility information.

The second development is a recent trend in routers to go beyond simple packet forwarding to do application-aware and security-aware routing. For example, some corporate routers are required to identify and provide QoS guarantees for SAP traffic. Such applications require the extraction of more packet fields which further increases the worst-case path length in the required parse tree.

We note that some routers also scan packets for strings and regular expressions. We do not consider searching for regular expressions as parsing in this paper. By parsing we mean the exploration of a *few* selected fields of a packet directed by a parse tree, as opposed to a scan of *all* payload bytes looking for a regular expression.

The third development is a trend among router vendors and academics to develop flexible and even re-programmable routers. Router vendors such as Cisco have increasingly made many components of their routers flexible such that the behavior of the component can be changed at run-time by changing parameters or even microcode. This is motivated by the need to respond to quickly changing customer demands in the face of long design cycles to design chips (2 years) and develop a router (3 years or greater). Flexibility has even reached the high-end with Cisco's CRS-1 [2] 92 Tbps router which uses 192 programmable packet processors. Ironically, for speed the CRS-1 uses a hardwired parser. Thus the CRS-1 cannot handle wire-speed implementations of new protocol formats as would be enabled by our Kangaroo system.

Academic researchers have recently advocated not only flexible but completely re-programmable routers to allow the creation of new routing and forwarding mechanisms. Efforts include Stanford's NetFPGA [21] and Washington University's Programmable Router [12]. Both efforts are motivated by the GENI project [7] that seeks to enable clean slate Internet architectures. But a programmable parser is an essential component of a programmable router in order to support new protocols easily without speed degradation. Thus we have implemented a high-speed, flexible parser for the NetFPGA platform we describe in Section VI.

In conclusion, there is a need for a flexible packet parsing module that can handle arbitrary protocol trees including trees with long worst-case paths containing a number of nested shim and protocol headers. Given that edge routers are routinely handling 10 Gbps, there is a need for at building such a flexible packet module that operates at close to 10 Gbps using say a FPGA, and at 40 Gbps to handle core router speeds using a small portion of a networking ASIC.

The contributions of this paper are: a new CAM-driven architecture to perform lookahead ((Section IV); a dynamic programming algorithm to select the amount of lookahead (Section V); and a prototype implementation on the Stanford NetFPGA board [21] (Section VII).

The rest of the paper is organized as follows. Section II defines the problem and presents a model of parser performance. Section III introduces lookahead and Section IV describes the Kangaroo architecture. Section V describes the algorithm that maps from protocol trees to CAM entries. Section VI describes a NetFPGA implementation, and Section VII describes the performance evaluation. Section VIII describes related work, and Section IX states conclusions.

II. Modeling Parsing Speed and Cost

In other network bottlenecks such as IP lookup, the worst-case is always a minimum sized (often 40 byte) packet. For parsing, however, the most time-consuming path in the parse tree is often caused by a packet larger than a minimum sized packet. Further, a path of 4 protocols with long headers (e.g. IP, 20 bytes) is easier to process than a path of 3 protocols with short headers (e.g., MPLS, 4 bytes). This is because the longer arrival time of the longer headers provide more processing "headroom".

Thus, we use a new measure for parsing speed: the minimum of the average number of bits per cycle across all paths in the parse tree. The average number of bits per cycle in a path is H/C, where H is the sum of the lengths of all protocol headers parsed along the path, and C is the number of cycles required to parse the path. Intuitively, a packet could arrive containing the headers corresponding to the path in time H/R, where R is a measure of the arrival rate. The packet will take C cycles to process. Wire speed processing occurs if the arrival time is greater than the processing time – in other words, if $H/R > C$. But that implies $H/C > R$. Thus using the smallest value of H/C formalizes the worst-case parsing time of a packet *relative* to the time it takes for the packet to arrive.

The number of parsing cycles required to parse a protocol path depends on the number of protocols parsed in a single cycle. We use the term *lookahead* to refer to the maximum number of protocols parsed in a parsing cycle. As an example, for commercial parse trees, the worst-case path is Ethernet, two 802.1Q, 4 MPLS, IP and ICMP, which has a total header length H of 496 bits. Using lookahead 1, the 9 protocols in this path will take 9 parsing cycles to process.

The speed measure for the worst-case ICMP path (after adding the lengths of all headers) will then be $496/9$ or 56 bits per parsing cycle. In most implementations, one parsing cycle may require 3-4 clock cycles; however, the data path can be pipelined so that 1 parsing cycle can be done every clock cycle. Thus the maximum throughput at 400 MHz is 18 Gbps. To achieve 40 Gbps, we can either use multiple copies of a lookahead 1 parser (packet parallelism) or process more than one protocol per parsing cycle (lookahead > 1).

Finally, our cost measure for a parser is the sum of the storage and logic costs expressed in gate equivalents. The storage costs will include costs to store the parse tree (small), and registers required to store the pipeline of packets that must be concurrently processed by the parse engine for speed. Note that packet parallel solutions will incur these costs for each replicated copy. We also consider a secondary cost measure, the total amount of power consumed. We will show that using lookahead > 1 provides a cheaper solution in terms of storage, logic, and power up to 40 Gbps.

III. Lookahead Challenges

We say that a parser has lookahead L if the parser processes up to L packet headers in a single cycle. Lookahead greater than 1 is impossible when parsing in software. It is, however, possible in hardware using multiplexers that can concurrently route several fields of a stored packet in memory to the parsing unit (Figure 5). A lookahead 2 parser can ideally have twice the throughput of a lookahead 1 parser. However, a lookahead implementation must deal with two challenges: dependencies and variable length headers.

Dependencies: Consider parsing the packet of Figure 2. If the first Ethernet type field (at an offset of 12 bytes) is 0x9100, there is a second Ethernet type Field (at an offset of 16 bytes). To do a lookahead of 2, the parser must simultaneously fetch two 16-bit fields corresponding to the two type fields at byte offsets 12 and 16. The difficulty, however, is that the value at byte offset 16 (second 802.1Q) is only *meaningful* if the first Ethernet type is 0x9100. By contrast, if the first Ethernet type field is IP, then the next offset the chip must fetch is the IP length field (at offset 116 bits). Dependencies can be broken by extracting *all* possible 2nd level offsets, and all possible 3rd level offsets. However if 6 protocols follow Ethernet and each protocol has an average of 5 possible successor protocols, a lookahead 3 chip has to fetch 30 possible fields in a cycle — a memory bandwidth challenge.

Variable Length Headers: A second challenge is lookahead across protocols whose header size is *variable* such as TCP. Both IP and TCP contain a 4-bit long header length (IHL) field in their headers. Alternately, in the GRE header, bits C, R, K and S are used to signal the presence of the checksum, routing, key, and sequence number fields respectively. Finally, MPLS headers are 32-bits long headers and can be chained. Bit 23 of the MPLS header is called the *stack bit*. The stack bit is set to 0 for all intermediate MPLS headers but is set to 1 for the final header.

Dealing with Dependencies: A uniform lookahead of 2 would require the chip to fetch fields corresponding to all 6 possible successor protocols of Ethernet (ARP, RARP, MPLS, IP, and two 802.1Q's). Uniform lookahead of 3 is even worse. However, the worst-case path we seek to optimize in Figure 3 is the long path on the right of 9 nodes (Ethernet, 2 802.1Q Headers, 4 MPLS Headers $\ldots$). There is no point doing lookahead for the path on the left (Ethernet, IP, TCP) because this path is only of length 3.

Thus, an efficient strategy for lookahead 3 is to only fetch the first Ethernet type field, together with speculative fetches of the second and third potential Ethernet type fields. No other fields are fetched. If the first Ethernet type field is IP, the speculative fetching of the next 2 potential type fields is wasted. But in that case the chip is traversing the easy leftmost path where lookahead is unnecessary.

Further, lookahead is not needed on portions of a long path containing long headers. For example, consider a protocol path of 9 protocols ending with IP and TCP that have 20 byte headers. Assume that the initial protocols (such as MPLS) have short 4 byte headers. To keep up with wire speed, while we may have to process 3 MPLS headers in 1 cycle (lookahead of 3 allows 12 bytes per cycle), we only need to process 1 IP or TCP header in 1 cycle (lookahead of 1 allows 20 bytes per cycle). We present a dynamic programming algorithm in Section V that calculates the minimum amount of lookahead. Figure 3 depicts *non-uniform lookahead* by partitioning each path into ovals of different sizes.

Dealing with Variable Lengths: Our main strategy is to avoid lookahead across variable length headers wherever possible. This works because many such headers are long (e.g., IP is 20 bytes), and we can use the ALU to calculate the offset of the next header. However, MPLS headers must be handled because they are 4-bytes long. It is easy to check if another MPLS header follows by checking the stack bit. However, to determine which protocol (IPv4, IPv6, or Ethernet) follows after the end of a sequence of MPLS headers, the parser must extract bits 23-36 following the MPLS headers to see if they are 4, 6, or 0 respectively. We have done a careful study of all variable length protocols including GRE and ARP (which we omit for lack of space) to prove that our methods apply to all existing protocols. Finally, for short headers that are variable length we add extra TCAM entries for each possible header length as explained in Section V.

IV. Lookahead Architecture in Kangaroo

For a lookahead of 3 in Figure 3, a parsing chip will simultaneously extract the Ethernet Type Field at an offset of 12 bytes (say Field 1), and two speculative type fields for potential 802.1q type fields (say Fields 2 and 3 at offsets 16 and 20 bytes respectively). The chip then executes the following decision logic:

```
if (Field 1 == 802.1q) AND (Field 2 ==
Inner 802.1q) AND (Field 3 == MPLS) then
    Continue parsing MPLS
else if (Field 1 == IP) then
    Continue parsing IP
end if
```

This decision logic corresponds to a longest prefix match on the concatenation of Fields 1, 2, and 3. But this can easily be done, as in IP lookups, by a Ternary Content Addressable Memory (TCAM) as in Figure 4, with one entry corresponding to the first decision and one entry (with wildcards for Fields 2 and 3) corresponding to the second decision.

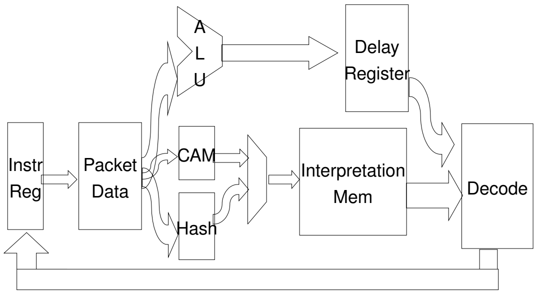

Fig. 4. Lookahead parsing architecture in the Kangaroo System.

If the chip merely extracted the first 24 bytes of the packet and used these 24 bytes to index the CAM, the CAM would be very wide (192 bits). Instead, our architecture concatenates the extracted fields before indexing, making the CAM word smaller. For example, in this example, we only need to index the CAM with three 16-bit fields or 48 bits. The architecture (Figure 5) allows a lookahead of k by extracting up to k 16-bit fields from anywhere within the first L bytes of the packet. We call this the Kangaroo system because it "leaps" over multiple headers in a single step. Such lookahead requires a CAM width of $k * 16$ bits. We use $k = 3$ and $L = 256$ in our implementation. A further reduction of TCAM entries can be achieved by adding an ALU to calculate lengths of variable length headers such as IPv4.

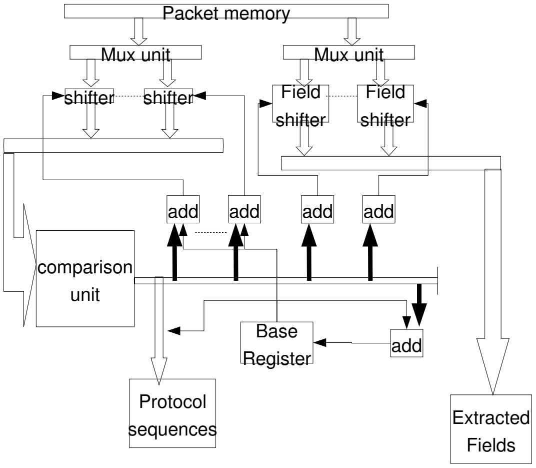

Fig. 5. A more detailed view of the architecture. Note that the comparison unit corresponds to the CAM, Hash Table and ALU in Figure 4.

Recall that even on long paths we wish to use lookahead of 1 on sequences of long headers (e.g. IP), and larger lookahead on sequences of short headers (e.g., MPLS). We facilitate this by using a hybrid comparison unit that consists of *both* a CAM and a hash table. Multiple fields of time-critical portions of a path are fetched via the CAM, while one field at a time can be fetched using the hash table for less critical portions. The choice of hashing versus CAM is made by a bit in the instruction corresponding to the last match.

The multiplexers that do field extraction must be designed carefully. A naive approach (for $k = 4$, $L = 256$) would route each of 256 * 8 bits to 64 outputs, requiring 256 * 8 * 64 wires. However, hierarchical multiplexing in two stages considerably reduces gate count. We describe details later.

Fields like IP can occur at various offsets, for example immediately after Ethernet or after MPLS. Without care, parsing IP would require a CAM entry for each possible offset. We avoid this duplication using a base register (Figure 5) that holds the packet offset of the last header parsed. CAM entries can then be stored *relative* to the base register. Each of these optimizations adds delay to the basic processing. However, these delays can be pipelined using 4 pipeline stages.

A. Detailed Architecture

Consider the parser architecture of Figure 5. An "instruction" requests a number of offsets from the packet memory and also specifies whether the comparison should be performed by the CAM or the hash table. The instruction also specifies the inputs of the ALU used to identify the end of variable length headers like IP. A multiplexer selects either the output of the CAM or the hash table to index the Interpretation Memory, in which each word is formatted as shown in Figure 6. Note that the architecture is fully flexible because all of the memories and CAMs can be configured to add support for any set of protocols.

The first part of the instruction word contains information about the currently recognized headers. The remainder of the word contains a bit that selects whether a hash or CAM match will be used next, the offset requests for the next execution cycle, and the inputs to the ALU. If the hash table is selected, the packet data is masked to remove unnecessary information. The bit is set by an offline algorithm that configures all memories as described in Section V.

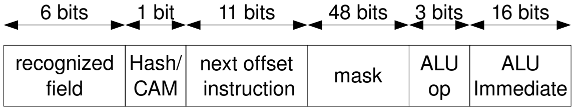

Fig. 6. The format of a ”parsing” instruction in the Interpretation Memory

The packet data module of Figure 4 uses shifters that can fetch any 16-bit long field of the first 256 bytes of the packet. The *Decode* module of Figure 4 consists of a *base register* that stores the offset of the last header that was parsed. Relative offsets are converted to absolute values by adding the value of the *base register*. Each protocol can specify fields to be extracted and stored (such as IP Destination Address) in an output FIFO as long the protocol is verified by the CAM.

In a single execution step, the outputs of the three shifters are concatenated to form a word that concurrently queries both the TCAM and the hash table. If the word matches, the corresponding index is used to fetch the “next instruction” (Figure 6). In our implementation, these instruction words are stored in a RAM indexed by the output of the TCAM. The multiple separate CAM nodes corresponding to each lookahead node are stored in a single shared CAM. To avoid ambiguity, each “CAM node” entry is prefixed with a small unique integer.

B. Sample Kangaroo Execution

We describe a sample Kangaroo execution parsing the packet shown in Figure 7, which is similar to that of Figure 2, with the exception of a *new* imaginary Layer 2 service protocol IMP. Assume that the IMP header is 4 bytes long and that it contains a 16-bit type field that starts at the beginning of the IMP header. If the IMP type field is $0x8847$, then the next protocol that follows is MPLS. The IMP protocol field follows an 802.1q header which also has a type field at the start of the field. The 802.1q type field is equal to $0x1234$ when IMP follows.

Assume 3 shifters numbered from 1 to 3. The CAM and hash table entries are shown in Figure 8. For illustrative purposes only, we ignore delays caused by the memory and pipeline registers and assume that the datapath of Figure 4 takes 1 *parsing* cycle. The parsing will vary between using lookahead 3 and lookahead 1. We only show the first 2 cycles due to lack of space.

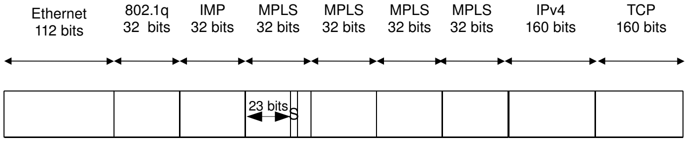

Fig. 7. The packet headers for an imaginary new service protocol we call IMP. S is the stack bit for MPLS.

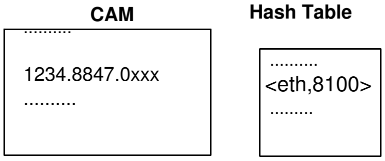

Fig. 8. The CAM and hash table contents for the example

cycle0: The parser uses lookahead 1 to extract the Ethernet field. shifter 1 contains the Ethernet type of the Ethernet field while shifters 2-3 are not used. The base register value is 0. The next match is specified to be in the hash table.

cycle1: This cycle recognizes the Ethernet type of 802.1Q, and sets up a lookahead 3 instruction to parse an 802.1Q header, an IMP header and one MPLS header. After a match with *Ethernet,8100* in the hash table, an instruction is fetched that recognizes 802.1Q and makes requests for 3 offsets: a) the offset of the Ethernet type of the 802.1q header (112 bits) b) the offset of the Ethernet type of the IMP header (112 + 32 bits, see Figure 7 and c) the bit sequence that starts from the stack bit (23rd bit in MPLS header) of the possible MPLS header that follows the Ethernet header. The next match is specified to be in the CAM. Thus, shifter1 is assigned bit offset 112, shifter2 is assigned offset 112 +32 and shifter3 is assigned offset 112 + 32 + 32 + 23. The base register becomes 112 which is where the Ethernet header ends. This will cause a match with the 1234.8847.0xxx entry of the CAM in the next cycle which means that the first Ethernet type field corresponds to IMP and IMP’s Ethernet type indicates that MPLS follows. Finally, since the MPLS stack bit is 0, more MPLS headers should be expected.

Further cycles (not shown) are needed to parse up to TCP. However, the sample execution shows how a new protocol IMP can be handled by our flexible parser in 2 cycles (instead of 4 cycles when implemented naively) by adding one entry to the CAM and hash table.

V. Compiling Parser Instructions from Parse Trees

Besides the hardware architecture, the second part of the Kangaroo system is an offline algorithm that “compiles” the parse tree in a few seconds to output the CAM and hash table entries whenever packet headers are changed. The parsing co-processor is then briefly (less than 1 msec) taken offline while the CAM and hash table entries are updated.

The offline algorithm needs a specification of a parse tree. The simplest specification of a parse tree is a graph with one vertex for each protocol. Each protocol node N has outgoing edges to the protocols that can appear after N. However, some protocols such as IPv4, require the examination of more than one 16-bit field in the header. Also, there can be cycles in such a graph. One can have a packet that has Ethernet over MPLS; then over the inner Ethernet one can have MPLS again, and so on. In practice, there are limits to such absurd recursion, but they must be specified.

Thus we specify a parse tree using a graph whose vertices denote the *fields* that need to be examined in a single step. For example, the graph (Figure 9) has two nodes (IPv4_1 and IPv4_2) for the IPv4 length and protocol fields respectively. We require that the graph be a Directed Acyclic Graph (DAG) to eliminate ambiguities caused by cycles. This is done by unrolling recursive traversals within the original graph up to the specified limits.

Variable length headers are modeled using dummy nodes that carry no information. However, the existence of dummy nodes in a path means that the fields of subsequent protocols should be fetched according to the size represented by each dummy node. The offline algorithm ensures that dummy nodes are always included in a group of other nodes, but TCAM values are not generated for dummy nodes. Figure 9 contains the graph which models a variable length IPv4 header using three dummy nodes (nodes containing a *) assuming that the IPv4 header has a length of either 5, 6, or 7 words.

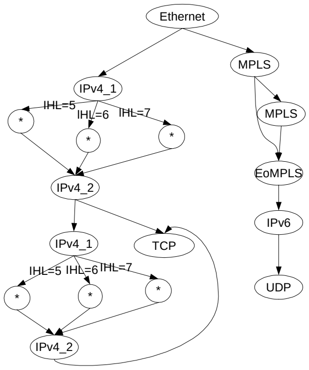

Fig. 9. Protocol graph. Note the three dummy nodes that follow IPv4_1 and the two nodes (IPv4_1 and IPv4_2) that represent the two fields of IPv4.

A. Algorithm to Create Table entries

We now describe an efficient dynamic programming algorithm that can be run offline to create the hash and CAM table entries that control parsing at run-time. In Section II we measured the speed of a parser path as H/C where H is the total length of headers in the path and C is the number of parsing cycles. The number of parsing cycles of a path is the number of lookahead clusters (see ovals in Figure 3) on a path. The goal of the algorithm is to decompose every path into lookahead clusters while minimizing CAM entries and maintaining the worst-case speed requirement. We assume that the major cost of the parsing engine is the the amount of TCAM; minimizing TCAM storage minimizes size and power.

Based on the desired bit rate and clock speed, the designer calculates the required bits per cycle B of the parse engine. For example, at 400 MHz and 40 Gbps, the chip must process 100 bits per clock cycle. Actually, what we have so far called a cycle is best referred to as a *parsing* cycle and actually takes 4 *clock* cycles: 2 clocks to read the interpretation memory and 2 to read the packet. However, by concurrently working on 4 packet headers, the required speed of B bits per clock cycle translates into B bits per parsing cycle.

Algorithm Specification: Given a directed acyclic graph $G = (V, E)$, each vertex of which is labeled with the size of the header it represents, a maximum lookahead of k elements per access, and a required speed B in bits/cycle, cluster the nodes of G into clusters such that **i)** each cluster contains no more than k connected nodes **ii)** for every path from the root to any leaf, $P/C \leq B$, where P is the number of header bits in the path and C is the number of clusters in the path and **iii)** the number of CAM entries is minimized

The algorithm uses dynamic programming, a form of recursion that avoids recomputing repeated subproblems. We specify the recursion by describing the initialization, the cost function, the base cases at which the recursion "bottoms out", and the recursive step.

Since the parse "tree" is a DAG, a starting node s can be found using a topological ordering of the DAG in $O(|V|+|E|)$ time. For each subproblem, the cost function is the quantity $OPT(i, b)$ that denotes the minimum number of CAM entries that are required from the subtree rooted at i subject to the speed requirement of b bits per cycle for the subtree. The algorithm output is *OPT (s, B)* where s is the start node of the graph. The base cases consist of all subtrees with up to k nodes rooted at some node v_t, in which every path from v_t ends at a leaf node, and in which none of the nodes of the path is connected with nodes that are not in the path.

The recursive step requires a notion called the *fringe* of a path. Formally, the fringe of a path is the set of all nodes that are not contained in the path but are neighbors of nodes in the path. The intent is to traverse a portion of the subtree in one step using lookahead k, and then to continue recursively with each node in the fringe of the first lookahead step. The cost of the original subproblem is the *sum* of the number of CAM entries required by each node in the fringe. CAM storage is minimized by picking the path with the minimum such value. More formally, the minimum number of entries for the subproblem starting at node i with speed requirement b (and initial speed constraint B) is shown in eq. (1)

In this formula, $Paths(i)$ refers to the set of paths that start from i such that the number of the vertices in each path does not exceed k. Also, $entries(p)$ is the sum of the number of outgoing edges of each node in path p. Finally, $W(p, j)$ is defined for a node j on the fringe of path p as follows: $W(p, j)$ is the sum of the header lengths of all nodes from the start of the path to the node on the path that is the predecessor of node j.

The algorithm recursively builds the solution starting from the root and proceeding until all leaves are covered. This expensive recursive solution is converted to an efficient dynamic programming version using memoization, by storing

$$OPT(i,b) = \min_{p \in Paths(i)} \{ \sum_{j \in Fringe(p)} entries(p) + OPT(j, B + (b - W(p,j))) \} \tag{1}$$

outcomes of recursive calls in an array. The complexity after memoization is $O(|E||V|d^k)$, where d is the maximum degree. Given that the lookahead k is small (< 3) and the branching factor d of real parse trees is also small (< 10), the algorithm takes only a few seconds to run.

VI. NetFPGA Kangaroo Implementation

We prototyped our lookahead architecture on the NetFPGA platform [21] which contains a Xilinx Virtex 2 pro FPGA. The parser can output two 32-bit fields for each of 6 protocols. The top-level design is a four stage pipeline. In the first stage, the base register is used to read data from the packet. The second stage masks out relevant bits from the fetched packet data. The third stage performs the lookup using the CAM or hash table. The fourth stage looks up the instruction memory. There is also an execution unit that executes the instruction but it is placed in the first pipeline stage as it uses only combinatorial logic.

Muxing of 256 bytes to three 16-bit fields is done efficiently in two steps using FPGA block RAMs. Consider a request for 16 bits starting at offset 325. The first step produces the addresses of two 16-bit words that are guaranteed to contain the requested bits. In the example, these are the words at positions 20 and 21 (found by dividing 325 by 16, implemented by removing the 4 least significant bits).

In the second step, the two 16-bit words output by the first step (say w_0 and w_1) and the 11 bit offset are used to produce the final result. This is done using 16 multiplexers whose inputs are arranged according to the following rule. Each bit i of the output can be any bit of w_0 that is in position greater than i and any bit of w_1 that is in position less than i. For example, the first bit of the output can be any bit of w_0. The second bit of the output can be any bit of w_0 not including the first bit or the first bit of w_2. By contrast, a single step multiplexer would have required 2048 * 16 * 128 gates and wires.

The design was implemented in Verilog and downloaded to the NetFPGA board. The logic utilization of the Virtex 2 pro /device 50 FPGA of the board was 10% and the speed that we could achieve after place and route was 70 MHz. Since the I/O capabilities of the NetFPGA are limited to 1 Gbps in each of the four network interfaces, we generated traffic internally to demonstrate higher parsing speeds. We used the NetFPGA so we could verify timing and execution in lieu of an ASIC. We monitored the outputs of the system using the Chipscope analyzer after connecting the Xilinx USB Cable to the JTAG port of the board.

Using a lookahead of 3, we were able to implement parsing in the NetFPGA at 10 Gbps using a clock of 63 MHz. The implementation took less than 10% of the available logic (2410 out of 23,616 slices) and less than 20% of the RAM blocks (51 out of 232). This leaves enough room to implement other processing tasks. As we will see, an implementation on an ASIC is even less resource intensive.

VII. Kangaroo Evaluation

We obtained the following parse tree supported by several Cisco routers. Shim headers that can follow Ethernet are 802.1q, nested 802.1q, resirc tag, service tag, 802.1ah and 802.1ad. Ethernet and Shim headers can be followed by up to 4 MPLS headers, ARP, RARP,IPv4, or IPv6. MPLS is followed by Ethernet,IPv4, or IPv6. IPv4/IPv6 is followed by TCP, UDP, GRE, ESP, ICMP, or a second IPv4 header. IPv6 can be followed by an IPv6 extension header which, in turn, is followed by TCP, UDP, ESP or ICMPv6. Finally, GRE can also be followed by IPv4/IPv6. The parse tree support up to three different lengths for IPv4, and up to eight different lengths for GRE. Note that although we have used a specific parse tree for our evaluation, the architecture can handle any protocol,

We compared our results against a simple programmable parser of lookahead 1 (the state of the art today) that parses a single protocol header in one step. Such an architecture can be obtained if we remove the CAM from the matching unit of Figure 5 and keep only the hash table. We verified this reference design by comparing it to the gate count and speed of a commercial programmable lookahead parser implemented by Cisco Systems.

a) Speedup of Lookahead: Figure 10 shows the maximum parsing speed in bits per cycle for five architectures: lookahead 1 without an ALU, lookahead 1 with an ALU, and lookahead 2, 3, and 4 with an ALU. The figure shows that parsing at 40 Gb/s can be done using lookahead 3 (96 bits per cycle) at around 400 MHz. From Figure 10, we can calculate the relative speedups of each architecture against the lookahead 1 without an ALU. The addition of an ALU speeds up lookahead 1 parsing by only 10%. Lookahead 2 improves throughput by more than 50%, and lookahead 3 almost doubles throughput compared to Lookahead 1. Further, lookahead 4 and higher is almost useless for the parse tree we used.

b) CAM cost of Lookahead: Figure 11 contains the number of bytes of TCAM for each architecture to achieve the rates shown in Figure 10. TCAM bytes is the number of TCAM entries multiplied by the number of TCAM bytes per entry. Note that a TCAM entry for lookahead 2 is 4 bytes, while an entry for lookahead 3 is 6 bytes. Both lookahead 2 and lookahead 3 require 40 CAM entries, but this is 160 bytes for lookahead 2 and 240 bytes for lookahead 3. The cost for lookahead 4 is 330 bytes with no speed gain.

A. Packet parallelism versus lookahead

Figure 12 shows the cost in FPGA slices for four different FPGA implementations. The first three bars starting from the

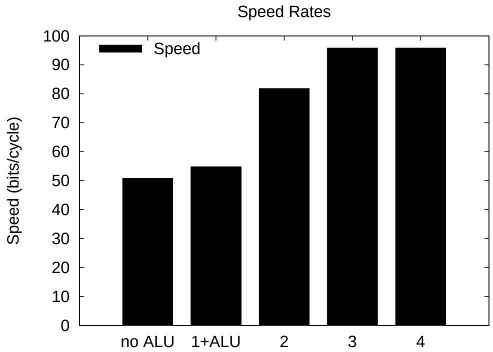

Fig. 10. The worst case speeds for various architectures. The bar labeled 2, for instance, represents lookahead 2.

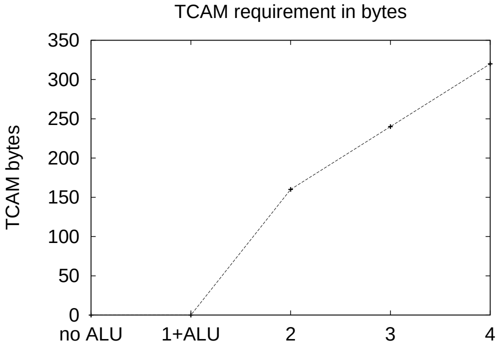

Fig. 11. TCAM storage costs in bytes for various lookahead values

left show the cost in slices for lookahead of 1, 2, and 3 respectively. The rightmost bar represents packet parallelism using two copies. It is obtained by taking twice the cost of the leftmost bar. The solid portion of each bar represents the logic costs excluding CAM costs, while the remainder represents CAM costs.

The comparison understates the cost of packet parallelism because it ignores the re-sequencing costs. Note also that the logic cost (excluding CAM costs) of lookahead 2 and 3 is about 20% higher than Lookahead 1 because of the extra cost of multiplexing.

In Figure 12, if we disregard the cost of the TCAM, the cost of a lookahead 3 implementation is 1770 slices while the cost of a 2-copy implementation of comparable speed (packet parallelism) is 2600 slices. Thus, disregarding FPGA CAM costs, lookahead is cheaper than packet parallelism. However, if we add the CAM costs, the cost of lookahead 3 is 2660 slices which is *more* than the 2-copy solution. However, this comparison is biased by the lack of native support for TCAMs in Virtex-2 Pro FPGAs. Even 40 CAM entries are expensive because TCAMs have to be naively implemented using logic. On the other hand, in an ASIC (as we show next), the cost of 40 TCAM entries is negligible. Thus the right comparison is indeed to disregard CAM costs.

c) Extension to ASICs: Networking ASICs being designed today use 65nm technology that corresponds to a density of 100k CAM Bits/square mm. Thus, 1 square mm of the ASIC can store 100K CAM bits. Using 48 bit wide CAMs, this is about 2000 CAM entries, which is much more than sufficient for the 40 entries needed for today's parse trees. Further, reasonably priced ASICs (smaller than $100) are around 180 square mm in size. Thus, the logic together with the CAM entries will take less than 1% of the chip area, which allows plenty of room for other logic such as lookups and classification. Given that a lookahead 3 implementation can achieve 96 bits/cycle, a 420 MHz implementation can provide a throughput of 40 Gbps.

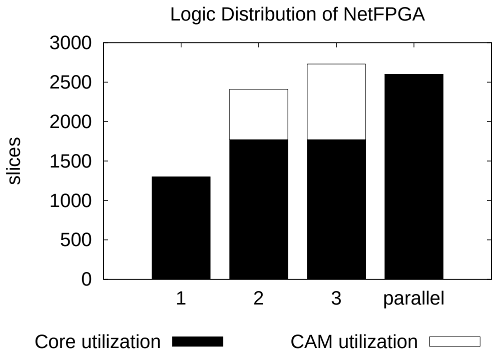

Fig. 12. The cost in FPGA logic slices for the various architectures. The bar labeled 2 for instance represents lookahead 2. The bar labeled parallel represents 2 copies of lookahead 1.

For comparison, we also used a Cisco Verilog implementation of a lookahead 1 parser. Compiled using an IBM ASIC library, the logic cost was 5K flops. Thus the total cost of a 2-copy packet parallel implementation (excluding re-sequencing costs) is at least 10,000 flops to reach 40 Gbps. Given that there is roughly a 6: 5 ratio between flops and TCAM bits (from the IBM ASIC library used by Cisco), the "cost" of 240 bytes of TCAM (for a lookahead 3 implementation) normalized to flops is 1600 flops. Assuming that the logic for lookahead 3 costs 20% more than lookahead 1 (see Figure 12), the total logic cost of lookahead 3 is 6000 flops (logic) + 1600 flops (CAMs) which is 7600 flops. Thus the cost of packet parallelism, even measured conservatively, is at least 30% more than that of lookahead. This confirms the intuition based on the FPGA results.

Further, for 65nm at 400 Hz, 5K flops consume 54mW while 320 bytes of TCAM consumes 4.8 mW. Thus the packet parallel solution uses almost double the power. The other costs of replication (packet storage, re-assembly logic) only strengthen the case for lookahead up to 40 Gbps.

Finally, we note that all programmable parsers will require at least 2 cycles to read an Interpretation Memory and 2 cycles to read the packet. Thus even a parser with lookahead 1 will need to be pipelined 4-deep to achieve 18 Gbps. Using two copies will only double storage. Thus, packet register storage using packet parallelism is also worse than a comparable lookahead 3 solution.

VIII. Related Work

Most prior work in efficient parsing is in the context of software and produced innovative parse trees representations including BPF [18], Pathfinder [14] and DPF [15]. While there is some prior work in programmable hardware parsing (e.g., [17]), we are the first to consider lookahead greater than 1. There are also patents by EZchip [6] and Avici [8] that use TCAMs for programmability but there is no evidence that they use *lookahead*, the main new idea in this paper. Additionally, none of the work we have seen in industry describes a compiler to automatically generate CAM entries.

Our offline algorithm has some analogs to the dynamic programming algorithm used in [20] to find the optimal traversal of a trie for multi-bit IP lookup. Non-uniform traversal, however, requires CAMs unlike [20]. Our dynamic programming algorithm is also more intricate because it must optimize across every possible fringe of a subtree and not just ”flat” fringes.

IX. Conclusions

While we have shown good speedups with lookahead, as with IP lookups, there are clearly other approaches to speeding up parsing. First, within each header some amount of parallelism can be exploited without using lookahead, as in extracting the IP Length and IP Protocol from the IP header. Second, the parse tree traversal can be pipelined without using lookahead. Third, one could replicate multiple slower parsers and use packet-by-packet parallelism. Each of these methods bears investigation and has their own set of tradeoffs.

The biggest advantage of Kangaroo’s lookahead parsing is its ability to deal cheaply with a large number of consecutive headers that appear in a small number of packet bytes. Figure 2 gives a real example which contains 6 headers (2 802.1q and 4 MPLS) in 24 bytes. As speeds increase, such worst-case sequences will greatly stress standard parsing algorithms that do not employ lookahead. A little lookahead can avoid the need for replicating the entire parsing logic to handle such sequences of small headers.

On the other hand, the disadvantage of lookahead parsing is that it requires the use of CAMs (which are less dense than SRAMs within ASICs or FPGAs) and it needs more complicated fetching of the packet data. However, a limited amount of lookahead parsing requires only a small number of CAM entries. Similarly, if we limit ourselves to fetching consecutive headers, the cost of the muxing becomes very cheap. The dynamic programming algorithm can be modified to calculate the best use of lookahead given both restrictions. Thus in this form, at the very least, we are confident that lookahead parsing will be a useful complement to other techniques.

The CAM-driven architecture we devised for Kangaroo may also be interesting in its own right. For example, it can be used to implement IP lookups and Packet Classification as well as parsing. While it is obvious that any architecture that has a Ternary CAM can implement lookups and classification, the use of a tree of CAM nodes may reduce memory compared to laying out the entire database in a CAM. A CAM-driven architecture, with suitable augmentation, may provide an interesting alternative for a network processor design.

As we look back to protocol developments in the last ten years, we see the emergence of a large number of shim headers that were designed to add information (such as tags, tunnel IDs) after the fact to protocols such as IP and Ethernet that were not designed for these purposes. There has also been an increasing trend to make finer granularity routing and QoS decisions based on more header fields. Finally, both research and market forces have led a trend towards more flexible routers, and hence towards flexible parsers. None of these trends show any signs of abating. Thus we believe that high-speed parsing will remain a fundamental bottleneck, as worthy of study as IP lookups, packet classification, or switching.

X. Acknowledgements

We would like to thank Flavio Bonomi, Tom Edsall, and Pere Monclus from Cisco Inc. for their help and support. We thank Brian Kantor for naming our system. This work was made possible by a grant from Cisco Systems and NSF grant CSR-0509546.

References

[1] http://www.intel.com/design/network/products/npfamily/ixp1200.htm.

[2] http://newsroom.cisco.com/dlls/partners/news/2004/pr_prod_06-09.html.

[3] http://newsroom.cisco.com/dlls/2004/hd_052504c.html.

[4] http://www.cisco.com/en/US/docs/ios/12_2sr/12_2srb/feature/guide/lsrmbrfc.html.

[5] http://www.juniper.net/techpubs/software/nog/nog-mpls/html/check-mpls-rsvp8.html.

[6] http://www.patentstorm.us/patents/6944168/description.html

[7] http://www.geni.net/

[8] http://www.google.com/patents?id=j8iXAAAAEBAJ&dq=Avici

[9] http://www.networksorcery.com/enp/topic/ipsuite.htm.

[10] Ieee 802.1q review. *http://en.wikipedia.org/wiki/IEEE_802.1Q*.

[11] *IEEE Std 802.1Q-2005*

[12] A proposed architecture for the geni backbone platform. *Washington University Technical Report, WUCSE-2006-14*.

[13] J. Allen et al, IBM Powernp network processor: Hardware software and applications. *IBM Journal of Research and Development*, May 2003.

[14] M. Bailey, et al PATHFINDER: A pattern-based packet classifier. *OSDI*, 1994.

[15] D. Engler and F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. *ACM SIGCOMM*, 1996.

[16] A. Gupta, A. Kumar, and R. Rastogi. Exploring the trade-off between label size and stack depth in MPLS routing. *IEEE INFOCOM*, 2003.

[17] P. Kobiersky, J. Korenek, L. Polack. Packet header analysis and field extraction for multigigabit networks. *DDECS*, pages 96–101, 2009.

[18] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. *USENIX Winter Conference*, pages 259–270, 1993.

[19] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching architecture, 2001.

[20] V. Srinivasan and G. Varghese. Faster IP lookups using controlled prefix expansion. *ACM Transactions on Computer Systems*, Feb 1999.

[21] J.Lockwood et al NetFPGA – an open platform for gigabit-rate network switching and routing. *IEEE International Conference on Microelectronic Systems Education*, 2007.