Design Principles for Packet Parsers

Glen Gibb[†], George Varghese[‡], Mark Horowitz[†], Nick McKeown[†]
[†]Stanford University [‡]Microsoft Research
{grg, horowitz, nickm}@stanford.edu varghese@microsoft.com

ABSTRACT

All network devices must parse packet headers to d how packets should be processed. A 64 × 10 Gb/s Eth switch must parse one billion packets per second to e fields used in forwarding decisions. Although a nec part of all switch hardware, very little has been writt parser design and the trade-offs between different desig it better to design one fast parser, or several slow pa What is the cost of making the parser reconfigurable field? What design decisions most impact power and

In this paper, we describe trade-offs in parser design, tify design principles for switch and router designers describe a parser generator that outputs synthesizabl ilog that is available for download. We show that i) p parsers today occupy about 1-2% of the chip, and ii) future packet parsers will need to be programmable only doubles the (already small) area needed.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Ne Architecture and Design—*Network Communications*

Keywords

Parsing; Design principles; Reconfigurable parsers

1. INTRODUCTION

Despite their variety, *every* network device examines in the packet headers to decide what to do with each p For example, a router examines the IP destination ad to decide where to send the packet next, and a firewall pares several fields against an access-control list to d whether to drop a packet.

The process of identifying and extracting the appro fields in a packet header is called *parsing* and is the su of this paper. Our main thesis is that packet parsing is bottleneck in high speed networks because of the con ity of packet headers, and design techniques for low-la streaming parsers are critical for all high speed netwo devices today.

Why is packet parsing challenging? The length and format of packets vary between networks and between packets. A basic common structure is one or more headers, a payload, and an optional trailer. At each step of encapsulation, an identifier included in the header identifies the type of data subsequent to the header. Figure 1 shows a simple example of a TCP packet.

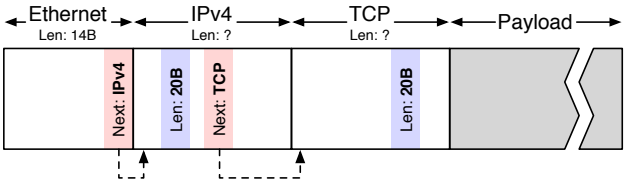


Figure 1: A TCP packet.

In practice, packets often contain many more headers. These extra headers carry information about higher level protocols (e.g., HTTP headers) or additional information that existing headers do not provide (e.g., VLANs[1] in a college campus, or MPLS[2] in a public Internet backbone). It is common for a packet to have eight or more different packet headers during its lifetime.

To parse a packet, a network device has to identify the headers in sequence before extracting and processing specific fields. A packet parser seems straightforward since it knows *a priori* which header types to expect.

In practice, designing a parser is quite challenging:

1. **Throughput.** Most parsers must run at line-rate, supporting continuous minimum-length back-to-back packets. A 10 Gb/s Ethernet link can deliver a new packet every 70 ns; a state-of-the-art Ethernet switch ASIC with 64 × 40 Gb/s ports must process a new packet every 270 ps.

2. **Sequential dependency.** Headers typically contain a field to identify the next header, suggesting sequential processing of each header in turn.

3. **Incomplete information.** Some headers do not identify the subsequent header type (e.g., MPLS) and it must be inferred by indexing into a lookup table or by speculatively processing the next header.

4. **Heterogeneity.** A network device must process many different header formats, which appear in a variety of orders and locations.

5. **Programmability.** Header formats may change after the parser has been designed due to a new standard, or because a network operator wants a custom header

[1]Virtual LAN [20].
[2]Multiprotocol Label Switching [19].

978-1-4799-1640-5/13/$31.00 ©2013 IEEE

field to identify traffic in the network. For example, PBB, VxLAN, NVGRE, STT, and OTV have all been proposed or ratified in the past five years.

While every network device contains a parser, very few papers have been published describing their design. We know of only two papers directly related to packet parsing [1, 8]: neither evaluated the trade-offs between area, speed, and power, and both introduce latency unsuitable for high speed applications. Regular expression matching work is not applicable: parsing processes a *subset* of each packet under the control of a parse graph, while regex matching scans the *entire* packet for matching expressions.

The goal of this paper is to educate designers as to how parser design decisions impact area, speed, and power via a design space exploration, considering both hard-coded and *reconfigurable* designs. We do not propose a single "ideal" design as we believe this is use-case specific.

A natural set of questions arise when setting out to design a parser. What are the area and power trade-offs between building one parser for an entire chip, versus breaking it down into multiple parallel parsers? What word width should we use to process the headers? A narrow word width requires a faster clock, while a wide word might require processing several headers in one step. How programmable should the parser be? What flexibility do users need to add arbitrary new headers, with fields in arbitrary locations?

We set about answering these questions as follows. First, we describe the parsing process in more detail (§2) and introduce parse graphs to represent header sequences and describe the parsing state machine (§3). Next, we discuss the design of fixed and programmable parsers (§4), and detail the generation of table entries for programmable designs (§5). Finally, we present a number of parser design principles identified through an analysis of different parser designs (§6). The designs were generated using a tool we built that, given a parse graph, generates parser designs parameterized by processing width and more. We generated over 500 different parsers and synthesized them against a TSMC 45 nm ASIC library. So as to compare our designs, each parser is designed to process packets in an Ethernet switching ASIC with 64 × 10 Gb/s ports.

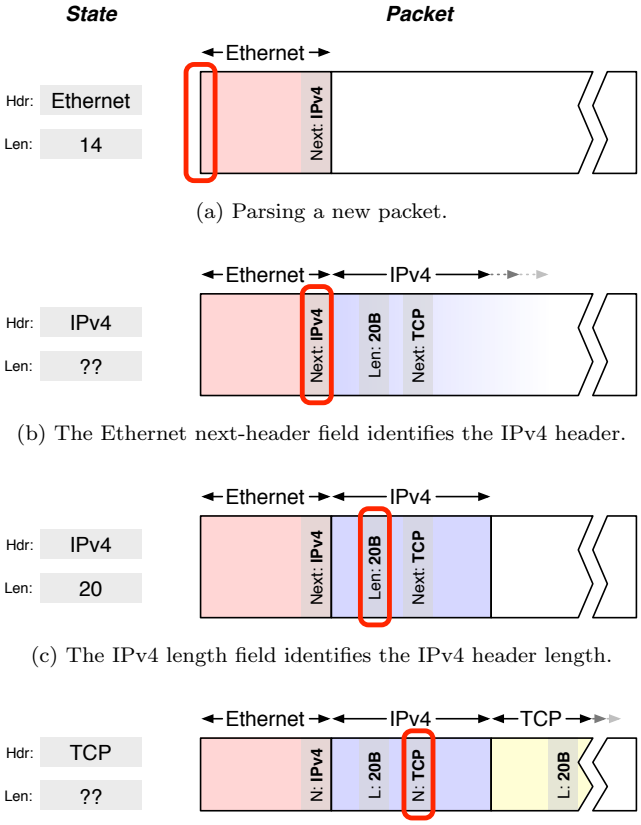
The contributions of this paper:

- We introduce the problem of designing streaming packet parsers (§4.5).
- We outline the similarities and differences to instruction decoding in processors (§4.6).
- We describe a new methodology for designing streaming parsers to optimize area, speed, and power (§6.1). By contrast, [8] used an FPGA implementation and a theoretical model that provides no insight into general ASIC area-power trade-offs.
- We describe multiple results, including those for a 64 × 10 Gb/s switch, an important component in today's networks, and identify design principles for switch designers (§6.2, §6.3).
- We show that a programmable parser occupies 1–2% of die area (§6.3).
- We show that the cost of programmability is a factor of two (§6.3).

2. PARSING

Parsing is the process of identifying headers and extracting fields for processing by subsequent stages of the device. Parsing is inherently sequential: each header is identified by the preceding header, requiring headers to be identified in sequence. An individual header contains insufficient information to uniquely identify its type. Next header fields are shown in Figure 1 for the Ethernet and IP headers—i.e., the Ethernet header indicates that an IP header follows.

The parsing process is illustrated in Figure 2. The large rectangle represents the packet being parsed and the smaller rounded rectangle represents the current processing location. Parser state tracks the current header type and length.



(a) Parsing a new packet.

(b) The Ethernet next-header field identifies the IPv4 header.

(c) The IPv4 length field identifies the IPv4 header length.

(d) The IPv4 next-header field identifies the TCP header.

Figure 2: The parsing process. (Field extraction omitted for simplicity.)

Processing begins at the head of the packet (2a). The initial header type is typically fixed for a given network and thus known by the parser (e.g., Ethernet). The structure of each header type is known by the parser, allowing it to identify the location of field(s) indicating the current header length and the next header type.

The parser reads the next header field and identifies the next header type as IPv4 (2b). IPv4 headers are variable length. The length is contained within a header field but it is initially unknown to the parser.

The IPv4 header length is read and the parser's state is updated appropriately (2c). The length identifies the next header's location and must be resolved *before* processing can commence on the subsequent header.

This process repeats until all headers are processed. Field extraction is performed in parallel with header identification; extraction is omitted from the diagram for simplicity.

3. PARSE GRAPHS

A *parse graph* expresses the header sequences recognized by a switch or seen within a network. Parse graphs are directed acyclic graphs, with vertices representing header types and edges specifying the sequential ordering of headers. Parse graphs for several use cases are shown in Figures 3a–3d.

The parse graph is the state machine that sequentially identifies the header sequence within a packet. Starting at the root node, state transitions are taken in response to next-header field values. The path that is traced through the parse graph matches the header sequence.

The parse graph (and hence state machine) within a parser may be either fixed (hard-coded) or programmable. A fixed parse graph is chosen at design-time and cannot be changed after manufacture, while a programmable parse graph is programmed at run-time.

Conventional parsers contain fixed parse graphs. To support as many use cases as possible, the chosen parse graph is a union of graphs from all expected use cases. Figure 3e is an example of the graph found within a switch—it is a union of graphs including those in 3a–3d. The union contains 28 nodes and 677 paths. This particular union is referred to as the "big-union" parse graph within the paper.

4. PARSER DESIGN

This section describes the basic design of parsers. It begins with an abstract parser model, describes fixed and programmable parsers, details requirements, and outlines differences from instruction decoding.

4.1 Abstract parser model

Recall that parsers identify headers and extract fields from packets. These operations can be logically split into separate blocks within the parser.

Extracted fields are used to perform forwarding table lookups in later stages of the switch. *All* input fields must be available prior to performing a table lookup. Fields are extracted as headers are processed, necessitating buffering of extracted fields until all required lookup fields are available.

Figure 4 presents an abstract model of a parser composed of *header identification*, *field extraction*, and *field buffer* modules. Header data is streamed into the parser and sent to the header identification and field extraction modules. The header identification module identifies headers and informs the field extraction module of header type and location information. The field extraction module extracts fields and sends them to the field buffer module. Finally the field buffer accumulates extracted fields, sending them to subsequent stages within the device once all fields are extracted.

Header identification

The header identification module implements the parse graph state machine (§3). Algorithm 1 details the parse graph walk that identifies the type and location of each header.

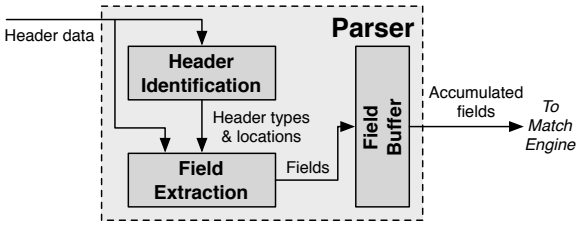


Figure 4: Abstract parser model.

Algorithm 1 Header type and length identification

```
procedure IdentifyHeaders(pkt)
    hdr ← initialType
    pos ← 0
    while hdr ≠ DONE do
        NotifyFieldExtraction(hdr, pos)
        len ← GetHdrLen(pkt, hdr, pos)
        hdr ← GetNextHdrType(pkt, hdr, pos)
        pos ← pos + len
    end while
end procedure
```

Field extraction

Field extraction is a simple process: chosen fields are extracted from identified headers. Extraction is driven by the header type and location information supplied to the module in conjunction with a list of fields to extract for each header type. Algorithm 2 describes the field extraction process.

Algorithm 2 Field extraction

```
procedure ExtractFields(pkt, hdr, hdrPos)
    fields ← GetFieldList(hdr)
    for (fieldPos, fieldLen) ← fields do
        Extract(pkt, hdrPos + fieldPos, fieldLen)
    end for
end procedure
```

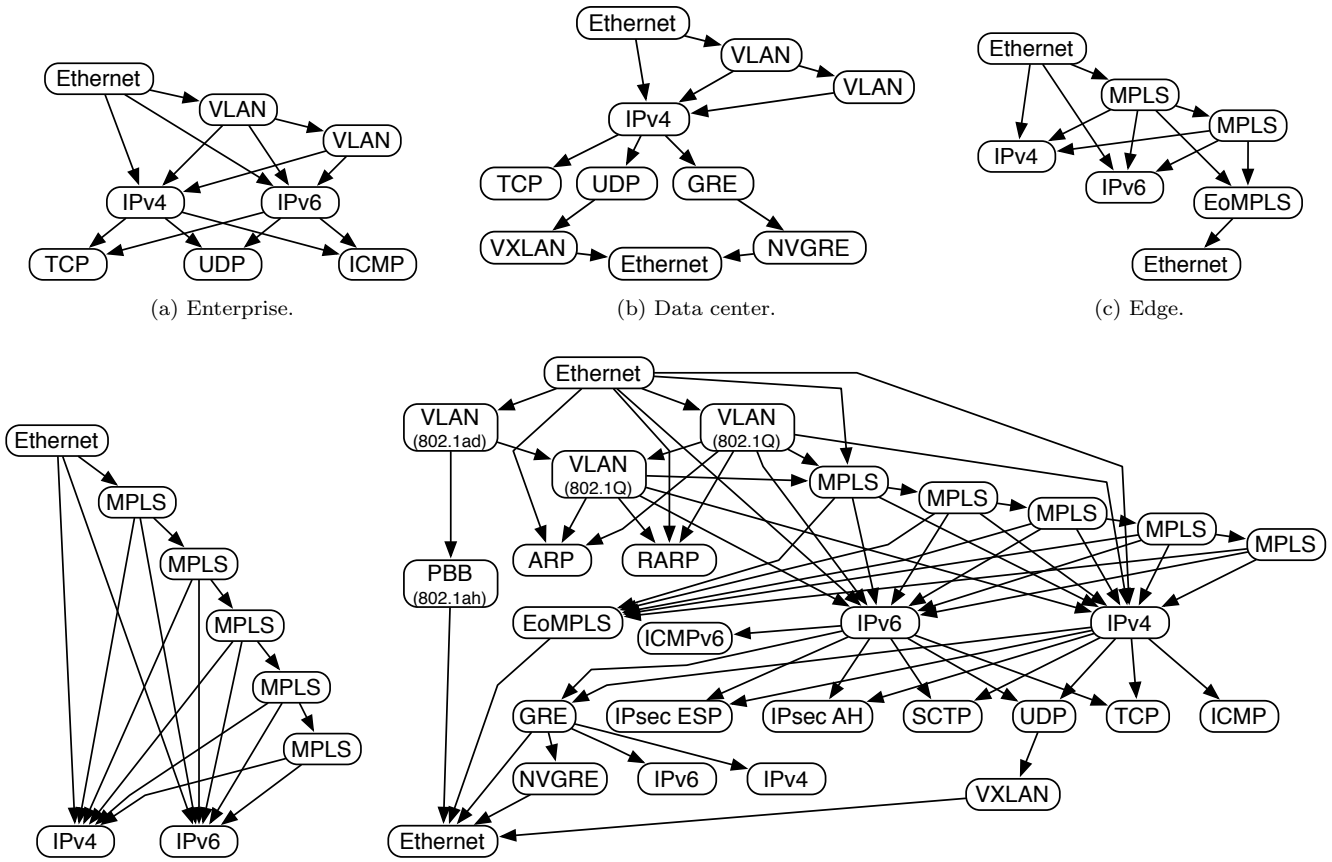
Field extraction may occur in parallel with header identification: extraction is performed on identified regions while identification is occurring on subsequent regions. No serial dependency exists between headers once the type and location are resolved, allowing fields to be extracted from multiple headers in parallel.

Field buffer

The field buffer accumulates extracted fields prior to table lookups by the switch. Extracted fields are output as a wide bit vector by the buffer as table lookups match on all fields in parallel. Outputting a wide bit vector requires the buffer to be implemented as a wide array of registers. One MUX is required per register to select between each field output from the field extraction block.

4.2 Fixed parser

A fixed parser processes a single parse graph chosen at design-time. The chosen parse graph is a union of the parse graphs for each use case the switch is designed to support. The logic within a fixed parser is optimized for the chosen parse graph.



(a) Enterprise. (b) Data center. (c) Edge.

(d) Service provider. (e) Union of use cases ("big-union").

Figure 3: Parse graph examples for various use cases.

The design we present isn't intended to precisely match the parser within any commercial switch, but it is qualitatively representative. As we show in §6.2, the area of a fixed parser is dominated by the field buffer. The width of this buffer is determined by the parse graph, and the need to send all extracted fields in parallel to the downstream match engine requires it be constructed from an array of registers and muxes. The lack of flexibility in the design of the buffer implies its size should be similar across parser designs.

4.2.1 Header identification

The header identification module is shown in Figure 5 and is composed of four elements: the state machine, a buffer, a series of header-specific processors, and a sequence resolution element.

The state machine implements the chosen parse graph, and the buffer stores received packet data prior to identification. One or more header-specific processors exist for *each* header type known by the parser—each processor reads the length and next-header fields for the given header type, identifying the length of the header and the subsequent header type.

A header identification block that identifies *one* header per cycle contains one header-specific processor per header type, and the sequence resolution element is a simple MUX. The MUX selects the processor output corresponding to the current header type. State machine transitions are determined by the MUX output.

Multiple headers per cycle can be identified by including multiple copies of some header processors. Each unique copy of a header processor processes data from different offsets within the buffer. For example, a VLAN tag is four bytes in length; including two VLAN processors allows two VLAN headers to be processed per cycle. The first VLAN processor processes data at offset 0, and the second VLAN processor processes data at offset 4.

At the beginning of a parsing cycle, the parser knows only the header type at offset zero—processing at non-zero offsets is *speculative*. At the end of a parsing cycle, the sequence resolution element resolves which speculative processors to use results from. The output from the processor at offset zero identifies the first speculative processor to use, and the output from the chosen first speculative processor identifies the second speculative processor to use, etc.

4.2.2 Field extraction

The field extraction module is shown in Figure 6. A buffer stores data while awaiting header identification. The fields to extract for each header type are stored in a table. A simple state machine manages the extraction process: waiting for header identification, looking up identified header types in the extract table, and extracting the specified fields from the packet.

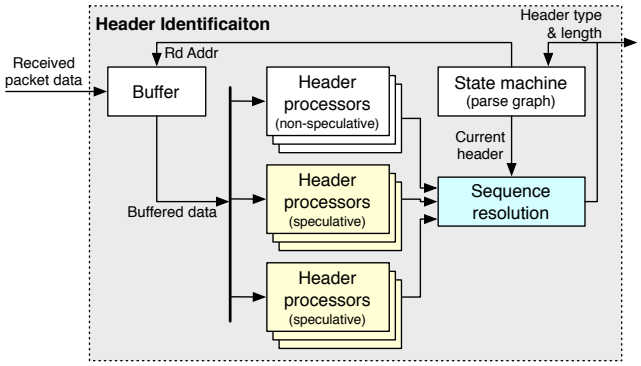


Figure 5: Header identification module (fixed parser).

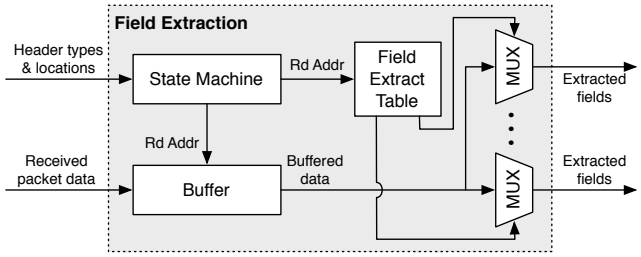


Figure 6: Field extraction module (fixed parser).

4.3 Programmable parser

A programmable parser uses a parse graph that is specified at run-time. We've chosen a state table driven approach for simplicity of understanding and implementation. State tables can be easily implemented in RAM and/or TCAM. Content-addressable memories (CAMs) are associative memories optimized for searching: all memory locations are searched in parallel for each input value. A binary CAM matches every bit precisely; a ternary CAM (TCAM) allows "don't-care" bits in entries that match any value.

The abstract parser model introduced earlier can be easily modified to include programmable state elements as shown in Figure 7. The key difference is the addition of the *TCAM* and *RAM* blocks. The TCAM stores bit sequences that identify headers. The RAM stores next state information, fields to extract, and any other data needed during parsing. The header identification and field extraction modules no longer contain hard-coded logic for specific header types, instead they utilize information from the two memories.

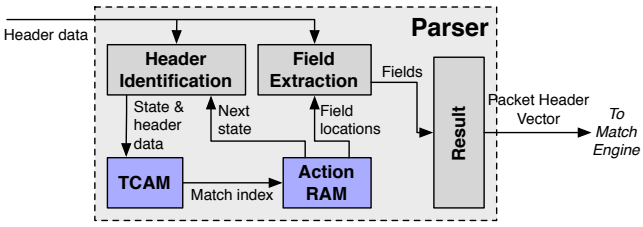


Figure 7: Programmable parser model.

The header identification module is simplified compared with a fixed parser, as shown in Figure 8. The module contains state tracking logic and a buffer; all header-specific logic moves to the TCAM and RAM. The current state and a subset of bytes from the buffer are sent to the TCAM, which returns the first matching entry. The bytes sent to the TCAM may be from contiguous or disjoint locations, with a total size between one byte and the whole packet, as determined by the specific design. The RAM entry corresponding to the matching TCAM entry is read and specifies the next state for the header identification module and the headers that were matched. The RAM entry may also specify data such as the number of bytes to advance and the subset of bytes to extract next.

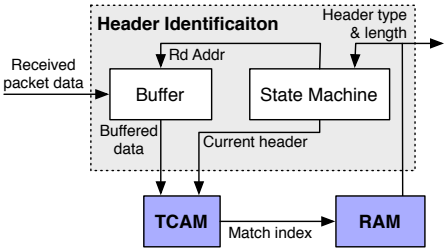


Figure 8: Header identification (programmable parser).

The field extraction module is almost identical to the fixed parser case, except the table containing field locations is moved out of the module and into the RAM.

4.4 Performance requirements

The parser must operate at line rate for worst-case traffic patterns in applications such as an Ethernet switch. Failure to parse at line rate causes packet loss when ingress buffers overflow.

A single parser instance may be incapable of operating at line rate—e.g., a parser running at 1 GHz in a 64 × 10 Gb/s switch must process an average of 640 bits per cycle. Instead, multiple parser instances may operate in parallel to achieve the required aggregate throughput, with each parser instance processing a different packet.

Many switches today operate with very aggressive packet ingress-to-egress latency targets. All elements of the switch, including the parser, must be designed to avoid excessive latency. This implies that packets should be parsed *as they are received* rather than buffering the entire packet (or header region) before commencing parsing.

4.5 Streaming vs. non-streaming operation

Parsers can be categorized as either streaming or non-streaming. A non-streaming parser receives the entire header sequence *before* commencing parsing, whereas a streaming parser parses *as headers are being received*. Kangaroo [8] is an example of a non-streaming parser.

Non-streaming parsers introduce latency as they wait to receive the header sequence, making them unsuitable for high-speed, low-latency networks. For example, buffering 125 bytes of headers at 1 Gb/s adds 1 µs of latency, which is particularly problematic in data center networks. The advantage of a non-streaming design is that it can access data anywhere within the header sequence during parsing, simplifying the design process.

Streaming parsers minimize latency by parsing as data is received. Data access is restricted to a small window of recently received data. The designer must ensure that each header is fully processed before it is flushed from the window.

The fixed and programmable designs above may be implemented as streaming or non-streaming parsers. Our focus is streaming implementations.

4.6 Comparison with Instruction Decoding

Packet parsing is similar to instruction decoding in modern CISC processors [15]. Instruction decoding transforms each CISC instruction into one or more RISC-like micro-operations (or μops).

Both are two-step processes with serial and non-serial phases. Parsing phases are header identification and field extraction; instruction decoding phases are instruction length decode (ILD) and instruction decode (ID).

ILD identifies each instruction's length. Instruction types are not required to decode lengths: a uniform structure is used across an instruction set, thereby allowing the same length identification operation to be used for all instructions. ILD is serial: the length of one instruction determines the start location of the next.

ID identifies the type of each instruction, extracts fields (operands), and outputs the appropriate sequence of μops. Multiple instructions can be processed in parallel once their start locations are known as no further decoding dependencies exist between instructions.

Despite the similarities, several important differences distinguish parsing from instruction decoding. Header types are heterogeneous: header formats differ far more than instruction formats. A header's type is identified by the preceding header as they do not encode their own type, unlike instructions.

5. PARSE TABLE GENERATION

A discussion of programmable parsers is incomplete without consideration of the parse table entry generation. In this section we describe parse table entries and present an algorithm and heuristics for minimizing the number of entries.

5.1 Parse table structure

The parse table is a state transition table. As the name implies, a state transition table specifies the transitions between states in a state machine. The table contains columns for the current state, the input values, the next state, and any output values. A table row specifies the state to transition to next for a specific current state and input values combination. A table row exists for each valid state and input value combination.

The parse table contains columns for the current header type, lookup values, next header type, the current header length, and the lookup offsets for use in the next cycle. Each edge in the parse graph is a state transition, thus each edge must be encoded as a parse table entry. Figure 9a shows a section of a parse graph, and Figure 9b shows the corresponding parse table entries.

Rather than using all packet data as input to the parse table, only fields indicating the header length and next header type are input. Inputing a subset of fields greatly reduces the table width (e.g., only four bytes are required from a 20B IPv4 header). The lookup offsets indicate which fields to use as inputs to the parse table.

5.2 Efficient table generation

The number of table entries can be reduced by encoding *multiple* transitions within a single entry, thereby identifying multiple headers. The parse graph fragment of Figure 9a can be encoded as shown in Figure 10a. This new table contains one less entry than the previous table for the same graph. Minimizing the entry count for a parse graph is beneficial since the state table is one of two major contributors to area (§6.3).

Combining multiple transitions within a table entry can be viewed as node clustering or merging. The cluster corresponding to the first entry in Figure 10a is shown in Figure 10b.

Table generation algorithm

Graph clustering is an NP-hard problem [5, p. 209]. Many approximations exist [3, 4, 7] but are poorly suited to this application.

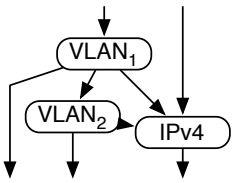
Kozanitis et al. [8] provide a dynamic programming algorithm to reduce table entries for Kangaroo. Kangaroo is a non-streaming parser that buffers all header data *before* commencing parsing; the algorithm is not designed for streaming parsers that parse data as it is received. The algorithm assumes it can access data anywhere within the header region, but a streaming parser can only use data from within a small sliding window.

We created an algorithm, derived from Kangaroo's algorithm, that is suitable for streaming parsers. Inputs to the algorithm are a directed acyclic graph $G = (V, E)$, the maximum number of lookup values k, the required speed B in bits/cycle, and the window size w. The algorithm clusters the nodes of G such that i) each cluster requires k or fewer lookups, ii) all lookups are contained within the window w, and iii) all paths consume a minimum of B bits/cycle on average, and iv) the number of clusters is minimized. Equation 1 shows the algorithm, which uses dynamic programming.

The algorithm is explained as follows. $OPT(n, b, o)$ returns the number of table entries required for the subgraph starting at node n, with a required processing rate of b bits/cycle, and with node n starting at offset o within the window. $Clusters(n, o)$ identifies all valid clusters starting at node n with the window offset o. The window offset determines the number of headers following n that fit within the window. *Clusters* uses the number of lookups k and the window size w to restrict the size of clusters. $Entries(c)$ returns the number of table entries required for the cluster c. $Successor(c)$ identifies all nodes reachable from the cluster c via a single edge. Figure 11a shows a cluster and the corresponding successors, and Figure 11b illustrates how the window w impacts cluster formation.

The recursive call to OPT identifies the number of table entries required for each successor node. Updated values are required for b and o. The updated b value reflects the number of bits consumed in the parsing cycle that processed c. The updated offset o reflects the amount of data that arrived and that was consumed while processing c.

Our algorithm is equivalent to Kangaroo's algorithm if the window size w is set to the maximum length of any header sequence. In this case, our algorithm can use any byte within the header region during any processing cycle.



(a) Graph fragment.

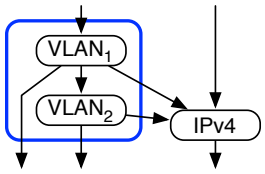
Curr. Hdr.	Lookup Val.	Next Hdr.	Hdr. Len	Next lookup offset
$VLAN_1$	0x0800	IPv4	4	0
$VLAN_1$	0x8100	$VLAN_2$	4	2
$VLAN_2$	0x0800	IPv4	4	0

(b) Corresponding parse table entries.

Figure 9: Sample parse table entries.

Curr. Hdr.	Lookup Vals.		Next Hdr.	Hdrs	Next lookup offset
$VLAN_1$	0x8100	0x0800	VLAN, IPv4	4	0, 6
$VLAN_1$	0x0800	--	IPv4	IPv4	0, 6

(a) Parse table entries using multiple lookups per entry.



(b) Clustered nodes in the first table entry.

Figure 10: Clustering nodes to reduce parse table entries.

$$OPT(n,b,o) = \min_{c \in Clusters(n,o)} \left(Entries(c) + \sum_{j \in Successor(c)} OPT(j, B + (b - W(c,j)), NewOffset(c,j,o)) \right) \quad (1)$$

Improving table generation

The algorithm described above (and Kangaroo's algorithm) correctly identifies the minimal set of table entries *only* when the parse graph is a tree. Each subgraph is processed independently by the algorithm, resulting in different sets of entries being generated for some overlapping sections of subgraphs. Figure 12a shows a parse graph fragment in which subgraphs rooted at nodes C and K share nodes F and G. Different sets of clusters are generated for the overlapping nodes for each subgraph, resulting in more entries than required. Figure 12b shows an alternate clustering in which a common cluster is generated for the shared region.

Suboptimal solutions occur only when multiple ingress edges lead to a node. Our heuristic to improve the solution removes the subgraph S with multiple ingress edges from the graph G, and solves S independently. A solution is then found for the graph $G - S$ and the results are merged. The merged solution is compared with the previous solution and is kept if it contains fewer edges. The comparison must be performed as solving the subgraph independently sometimes increases the number of edges. This process is repeated for each subgraph with multiple ingress edges. Figure 12c shows this process on a single subgraph.

6. DESIGN PRINCIPLES

A designer faces choices when designing a parser. Should the parser be constructed from many slow instances or a few fast instances? How many bytes per cycle should the parser process? What is the cost of including a particular header in the parse graph?

This section explores important parser design choices and presents principles that guide the selection of those parameters. We begin by describing a *parser generator* that allows exploration of the design space. We then present a number of design principles that we identified through our design space exploration.

Each parser presented in this section has an *aggregate* throughput of 640 Gb/s unless otherwise stated. 640 Gb/s corresponds to switch chips available today with 64×10 Gb/s ports [2, 6, 10, 11]. Multiple parser instances are required to achieve this aggregate.

6.1 Parser generator

A thorough exploration of the design space requires the comparison and analysis of many unique parser instances. To facilitate this, we built a parser generator that generates unique parser instances for user-supplied parse graphs and associated parameter sets.

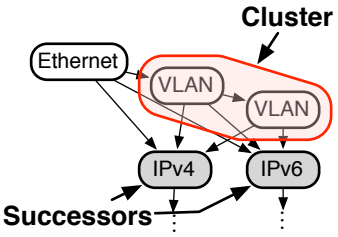
Our parser generator is built using the Genesis [14] chip generator. Genesis is a tool that generates design instances using an architectural "template" and a set of application configuration parameters. Templates consist of Verilog and Perl—the Perl codifies design choices to programmatically generate Verilog code.

Our generator produces fixed and programmable parsers as described in §4.2 and §4.3. Generator parameters include the parse graph, the processing width, the type (fixed or programmable), the field buffer depth, and the size of the programmable TCAM/RAM. The generator outputs synthesizable Verilog. All area and power results were generated using Synopsys Design Compiler G-2012.06 and a 45 nm TSMC library.

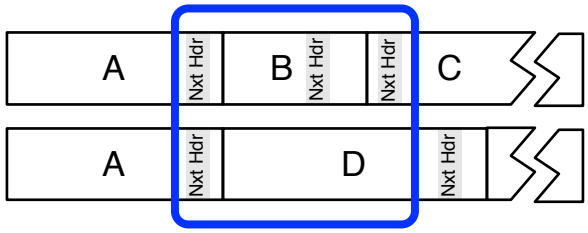
The parser generator is available for download from: `http://yuba.stanford.edu/~grg/parser.html`

Generator operation

Fixed parser: Two parameters are of particular importance when generating a fixed parser: the parse graph and the processing width. The parse graph is specified as a text

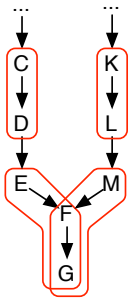


(a) Cluster and associated successors.

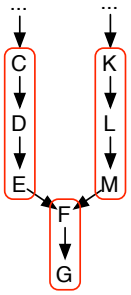


(b) The window (blue) restricts cluster formation. Next-header fields must lie within the same window: $\{A, B, C\}$ is a valid cluster but $\{A, D\}$ is not.

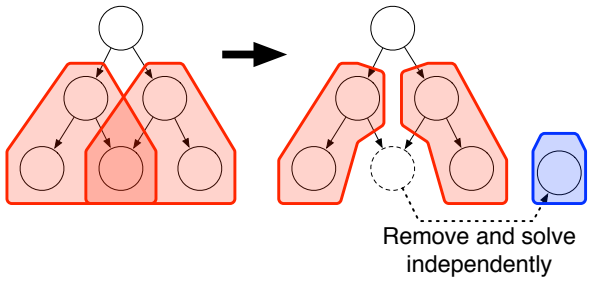
Figure 11: Cluster formation.



(a) Multiple clusters contain nodes FG.



(b) An alternate cluster construction places FG in *one* cluster.



(c) Subgraphs with parents are removed and solved independently to prevent nodes appearing in multiple clusters.

Figure 12: Improving cluster formation.

file containing a description of each header type. For each header type, the description contains the name and size of all fields, an indication of which fields to extract, a mapping between field values and next header types, and for variable-length headers, a mapping that identifies header length from field values. The IPv4 header description is shown in Figure 13.

```
ipv4 {
   fields {
      version        :  4,
      ihl            :  4,
      diffserv       :  8 : extract,
      totalLen       : 16,
      identification : 16,
      flags          :  3 : extract,
      fragOffset     : 13,
      ttl            :  8 : extract,
      protocol       :  8 : extract,
      hdrChecksum    : 16,
      srcAddr        : 32 : extract,
      dstAddr        : 32 : extract,
      options        : *,
   }
   next_header = map(fragOffset, protocol) {
       1 : icmp,
       6 : tcp,
      17 : udp,
   }
   length = ihl * 4 * 8
   max_length = 256
}
```

Figure 13: IPv4 header description.

Header-specific processors (§4.2.1) are created by the generator for each header type. Processors are fairly simple: they extract and map the lookup fields that identify length and next header type. Lookup field extraction is performed by counting words from the beginning of the header; next header type and length are identified by matching against a set of patterns.

The generator partitions the parse graph using the target processing width, as shown in Figure 14. Each region indicates headers that may be seen within a single cycle. Either one or two VLAN tags may been seen in the shaded region in the diagram. Header processors are instantiated at the appropriate offsets for each identified region.

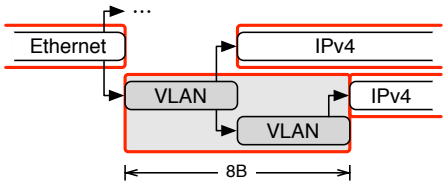


Figure 14: Parse graph partitioned into processing regions (red).

Header processing may be deferred to the subsequent region by the generator to minimize the offsets for a header. Referring to Figure 14, the shaded region could have included the first four bytes of the upper IPv4 header. However, the parser would then have required two IPv4 proces-

sors: one at offset 0 for the path VLAN → IPv4, and one at offset 4 for the path VLAN → VLAN → IPv4.

The field extract table (§4.2.2) is generated using the fields tagged for extraction in the parse graph description. A table entry for the IPv4 header would indicate to extract bytes 1, 6, 8, 9, and 12–19. The field buffer is sized to accommodate all fields that require extraction.

Programmable parser: Parameters important in generating a programmable parser include the processing width, the parse table dimensions, the window size, and the number and size of parse table lookups. A parse graph is not required to generate a programmable parser since the graph is specified at run-time.

Parameters are used by the generator to determine component sizes and counts. For example, the window size determines the input buffer depth within the header identification component and the number of mux inputs used when extracting fields for lookup in the parse table. Similarly, the number of parse table inputs determines the number of muxes required to extract inputs. Unlike the fixed parser, the programmable parser does not contain any logic specific to a particular parse graph.

The generator does *not* generate the TCAM and RAM used by the parser state table. A vendor-supplied memory generator must be used to generate memories for the process technology in use. (Non-synthesizable models are produced by the parser generator for use in simulation.)

Test bench: The generator produces test benches to verify the parsers it produces. The parse graph is used to generate input packets and to verify the output header vectors. The processing width determines the input width of the packet byte sequences.

6.2 Fixed parser design principles

Our design space exploration reveals that relatively few design choices have an appreciable impact on parser design—most design choices have a small impact on properties such as size and power.

Principle: The processing width of a single parser instance trades area for power.

A single parser instance's throughput is determined by $r = w \times f$, where r is the rate or throughput, w is the processing width, and f is the clock frequency. If the parser throughput is fixed, then $w \propto 1/f$.

Figure 15a shows the area and power of a single parser instance with a throughput of 10 Gb/s. Increasing the processing width increases area as additional resources are required to process the additional data. Power decreases because the clock frequency decreases (and decreases faster than the amount of logic increases).

Additional resources are required for two reasons. First, the packet data bus increases in width, requiring more wires, registers, muxes, etc. Second, additional headers can occur within a single processing region (§6.1), requiring more header-specific processor instances for speculative processing.

Principle: Use fewer faster parsers when aggregating parser instances.

Figure 15b shows area and power of parser instances of varying rates aggregated to achieve 640 Gb/s. The graph shows a small power advantage of using fewer faster parser over many slower parsers to achieve the desired aggregate throughput. Total area is largely unaffected by the instance breakdown.

The rate of a single parser instance does not scale indefinitely. Area and power both increase (not shown) as we approach the maximum rate of a single instance.

Principle: Field buffers dominate area.

Figure 15c shows parser area by functional block for several parse graphs. Field buffers dominate parser area, accounting for approximately two-thirds of total area.

There is little flexibility in the design of the field buffer: it must be built from an array of registers to allow extracted fields to be sent in parallel to downstream components (§4.1).

Principle: A parse graph's extracted bit count determines parser area (for a fixed processing width).

Figure 15d shows the total extracted bit count versus the parser area for several parse graphs. All points lie very close to the trend line. An additional data point is included for the simple parse graph of Figure 15e. This graph consists of three nodes but extracts the same number of bits as the big-union graph. The data point lies slightly below that of the big-union graph—the small difference is a result of slightly simpler header identification and field extraction logic.

This principle follows from the previous principle: the number of extracted bits determines the field buffer depth, and the field buffer dominates total parser area, thus the number of extracted bits should approximately determine the total parser area.

6.3 Programmable parser design principles

The fixed parser design principles apply, with several additional principles outlined below.

Principle: Parser state table and field buffer area are the same order of magnitude.

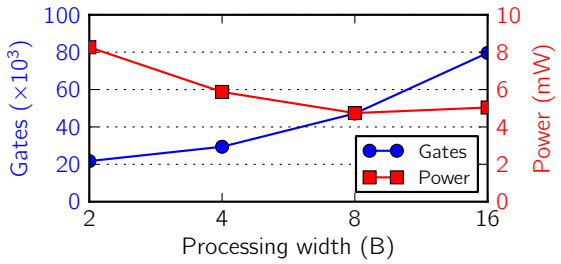
Figure 15f shows an area comparison between a fixed and a programmable parser. The fixed design implements the big-union parse graph. Both parsers include 4 Kb field buffers for comparison, and the programmable parser includes a 256 × 40 b TCAM. (Lookups consist of an 8 b state value and 2 × 16 b header fields.) The graph reveals that the programmable design is almost twice the area of the fixed design.

It is important to note that a fixed parser is sized to accommodate the chosen parse graph, while a programmable parser must be sized to accommodate all expected parse graphs. Many resources are likely to be unused when implementing a simple parse graph using the programmable parser. For example, the enterprise parse graph requires only 672 b of the 4 Kb field buffer.

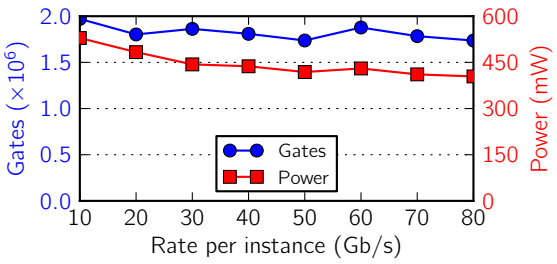
The 4 Kb field buffer and the 256 × 40 b TCAM are more than sufficient to implement all tested parse graphs. The TCAM and the field buffer are twice the size required to implement the big-union parse graph.

Observation: Programmability costs $1.5 - 3\times$.

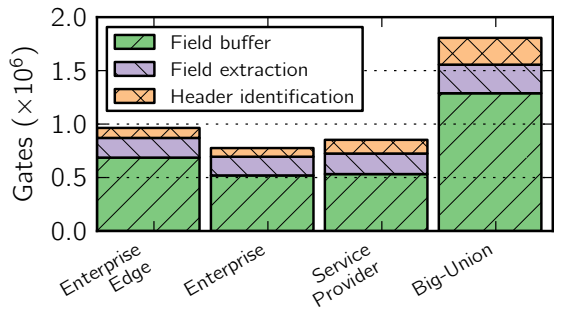
Figure 15f shows one data point, but comparisons across a range of parser state table and field buffer sizes reveals that programmable parsers cost between 1.5 and 3 times the area of a fixed parser (for reasonable choices of table/buffer sizes).



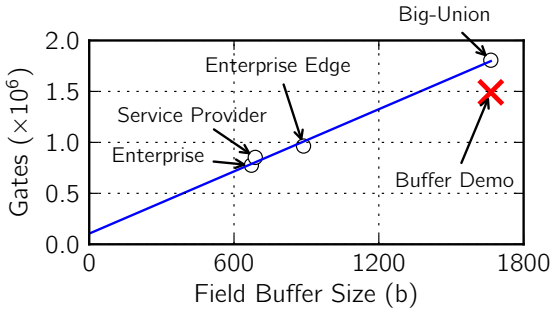
(a) Area and power requirements for a single parser instance. (Throughput: 10 Gb/s.)



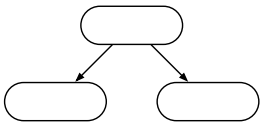
(b) Area and power requirements using multiple parser instances. (Total throughput: 640 Gb/s.)



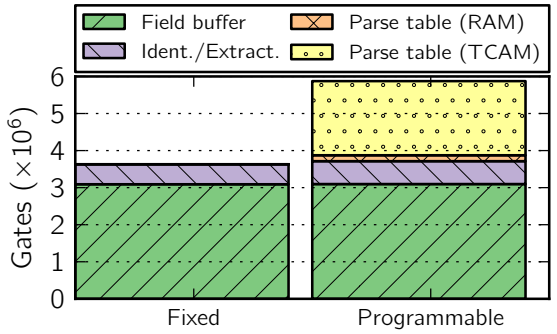
(c) Area contributions of each component for several parse graphs. (Total throughput: 640 Gb/s.)



(d) Area vs. field buffer size. Blue line shows linear fit for previously identified parse graphs. Red X is data point parse graph of Fig. 15e. (Total throughput: 640 Gb/s.)



(e) Simple parse graph that extracts the same total bit count as the Big-Union graph.



(f) Area comparison between fixed and programmable parsers. Resources are sized equally for comparison when possible. (Total throughput: 640 Gb/s.)

Figure 15: Design principles revealed via area and power graphs.

Observation: A programmable parser occupies 2% of die area.

Parsers occupy a small fraction of the switch chip. The fixed parser of Figure 15f occupies 2.6 mm^2, while the programmable parser occupies 4.4 mm^2 in a 45 nm technology. A 64 × 10 Gb/s switch die is typically 200 − 400 mm^2 today[3].

Principle: Minimize the number of parse table lookup inputs.

Increasing the number of parse table lookup inputs allows more headers to be identified per cycle, potentially decreasing the total number of table entries. However, the cost of an additional lookup is paid by *every* entry in the table, regardless of the number of lookups required by the entry.

Table 1 shows the required number of table entries and the total table size for differing numbers of 16 b lookup inputs with the big-union parse graph. The total number of table entries reduces slightly when moving from one to three lookups, but the total size of the table increases greatly. The number of table lookups should therefore be minimized to reduce total parser area since the parse state table is one of the two main contributors to area.

In this example, the number of parse table entries *increases* when the number of lookups exceeds three. This

[3]Source: private correspondence with switch chip vendors.

		TCAM	
Inputs	Entry count	Width (b)	Size (b)
1	113	24	2,712
2	105	40	4,200
3	99	56	5,544
4	102	72	7,344

Table 1: Parse table entry count and TCAM size.

is an artifact caused by the heuristic to reduce the number of table entries. The heuristic considers each subgraph with multiple ingress edges in turn. The decision to remove a subgraph may impact the solution of a later subgraph. In this instance, the sequence of choices made when performing three lookups per cycle performs better than the choices made when performing four lookups per cycle.

Our exploration reveals that two 16 b lookup values provides a good balance between parse state table size and the ability to maintain line rate for a wide array of header types. All common headers in use today are a minimum of four bytes, with most also being a multiple of four bytes. Most 4 B headers contain only a single lookup value, allowing two 4 B headers to be identified in a single cycle. We don't expect to see many headers shorter than four bytes in the future since little information could be carried by such headers.

7. RELATED WORK

Kangaroo [8] is a programmable parser that parses multiple headers per cycle. Kangaroo buffers all header data *before* parsing which introduces latencies that are too large for switches today. Attig [1] presents a language for describing header sequences, together with an FPGA parser design and compiler. Switch chips are ASICs not FPGAs, leading to a different set of design choices. Neither work explores design trade-offs or extract general parser design principles.

Much has been written about hardware-accelerated regular expression engines (e.g., [12,16,17]) and application-layer parsers (e.g., [13,18]). Parsing is the exploration of a small section of a packet directed by a parse graph, while regular expression matching scans all bytes looking for regular expressions. Differences in the data regions under consideration, the items to be found, and the performance requirements lead to considerably different design decisions. Application-layer parsing frequently involves regular expression matching.

Software parser performance can be improved via the use of a streamlined fast path and a full slow path [9]. The fast path processes the majority of input data, with the slow path activated only for infrequently seen input data. This technique isn't applicable to hardware parser design since switches must guarantee line rate performance for worst-case traffic patterns; software parsers do not make such guarantees.

8. CONCLUSION

The goal of this paper was to understand how parsers should be designed. To aid in this, we built a parser generator capable of producing fixed and programmable parsers, which we used to generate and study over 500 different parsers. A number of design principles were identified after studying the trade-offs between the generated parsers.

Our study reveals that area is dominated by the field buffer for fixed parsers, and the field buffer and parse table combined for programmable parsers. Multiple parsers can be aggregated to achieve a higher throughput: a small number of fast parsers provides a small power benefit (but no area benefit) over a large number of slow parsers. Programmability can be provided inexpensively: programmability doubles parser area, equating to an increase in total chip area from 1% to 2%.

9. REFERENCES

[1] M. Attig and G. Brebner. 400 Gb/s Programmable Packet Parsing on a Single FPGA. In *Proc. ANCS '11*, pages 12–23, 2011.

[2] Broadcom Trident II Ethernet Switch Series. http://www.broadcom.com/products/Switching/Enterprise/BCM56850-Series.

[3] G. Even, J. Naor, S. Rao, and B. Schieber. Fast approximate graph partitioning algorithms. *SIAM Journal on Computing*, 28(6):2187–2214, 1999.

[4] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proc. DAC '82*, pages 175–181, 1982.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[6] Intel Ethernet Switch Silicon FM6000. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf.

[7] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal*, (49):291âĂŞ–307, 1970.

[8] C. Kozanitis, J. Huber, S. Singh, and G. Varghese. Leaping Multiple Headers in a Single Bound: Wire-Speed Parsing Using the Kangaroo System. In *Proc. INFOCOM 2010*, pages 1–9, Mar. 2010.

[9] S. Kumar. *Acceleration of Network Processing Algorithms*. PhD thesis, Washington University, 2008.

[10] Marvell Prestera CX. https://origin-www.marvell.com/switching/prestera-cx/.

[11] Mellanox SwitxhX-2. http://www.mellanox.com/sdn/.

[12] J. Moscola, Y. Cho, and J. Lockwood. A Scalable Hybrid Regular Expression Pattern Matcher. In *Proc. FCCM '06.*, pages 337–338, 2006.

[13] J. Moscola, Y. Cho, and J. Lockwood. Hardware-Accelerated Parser for Extraction of Metadata in Semantic Network Content. In *IEEE Aerospace Conference '07*, pages 1–8, 2007.

[14] O. Shacham, O. Azizi, M. Wachs, W. Qadeer, Z. Asgar, K. Kelley, J. Stevenson, S. Richardson, M. Horowitz, B. Lee, A. Solomatnikov, and A. Firoozshahian. Rethinking Digital Design: Why Design Must Change. *IEEE Micro*, 30(6):9–24, Nov.-Dec. 2010.

[15] J. P. Shen and M. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill, 2005.

[16] J. Van Lunteren. High-Performance Pattern-Matching for Intrusion Detection. In *Proc. INFOCOM '06*, pages 1–13, 2006.

[17] J. Van Lunteren and A. Guanella. Hardware-accelerated regular expression matching at multiple tens of Gb/s. In *Proc. INFOCOM '12*, pages 1737–1745, 2012.

[18] S.-D. Wang, C.-W. Chen, M. Pan, and C.-H. Hsu. Hardware accelerated XML parsers with well form checkers and abstract classification tables. In *International Computer Symposium (ICS) '10*, pages 467–473, 2010.

[19] Wikipedia. Multiprotocol Label Switching. http://en.wikipedia.org/wiki/MPLS.

[20] Wikipedia. Virtual LAN. http://en.wikipedia.org/wiki/VLAN.