2011 Seventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems

400 Gb/s Programmable Packet Parsing on a Single FPGA

Michael Attig and Gordon Brebner
Xilinx Labs
2100 Logic Drive
San Jose, CA 95124, USA
(+1) 408 559-7778
{mike.attig,gordon.brebner}@xilinx.com

ABSTRACT

Packet parsing is necessary at all points in the modern networking infrastructure, to support packet classification and security functions, as well as for protocol implementation. Increasingly high line rates call for advanced hardware packet processing solutions, while increasing rates of change call for high-level programmability of these solutions. This paper presents an approach for harnessing modern Field Programmable Gate Array (FPGA) devices, which are a natural technology for implementing the necessary high-speed programmable packet processing. The paper introduces PP: a simple high-level language for describing packet parsing algorithms in an implementation-independent manner. It demonstrates that this language can be compiled to give high-speed FPGA-based packet parsers that can be integrated alongside other packet processing components to build network nodes. Compilation involves generating virtual processing architectures tailored to specific packet parsing requirements. Scalability of these architectures allows parsing at line rates from 1 to 400 Gb/s as required in different network contexts. Run-time programmability of these architectures allows dynamic updating of parsing algorithms during operation in the field. Implementation results show that programmable packet parsing of 600 million small packets per second can be supported on a single Xilinx Virtex-7 FPGA device handling a 400 Gb/s line rate.

Categories and Subject Descriptors

C.2.6 [Computer-Communication Networks]: Internetworking—Routers; C.1.3 [Other Architecture Styles]: Pipeline Processors; D.3.4 [Programming Languages]: Processors – Parsers.

General Terms

Algorithms, Performance, Design, Languages.

Keywords

High-speed packet processing. FPGA-based parallel processing. Domain-specific languages and compilers.

1. INTRODUCTION

As the Internet evolves, there is a growing need for non-trivial packet parsing at all points in the networking infrastructure, including the core carrier networks. Parsing is central to packet classification in order to identify flows and implement quality of service goals. Increasingly, it is also important to guide deeper packet inspection in order to implement security policies. Of course, packet parsing also continues to have a central role in the implementation of end-to-end communication protocols. With core networks moving to 100 Gb/s rates, and 400(±100) Gb/s rates on the horizon, packet parsing at line rates poses a major problem. A further complication is that parsing requirements can change frequently as network traffic patterns evolve and protocols are introduced, modified or replaced. This demands dynamic flexibility within networking equipment.

A packet in transit consists of a stack of headers, a data payload, and – optionally – a stack of trailers. At an end system, a packet might begin with a stack of Ethernet, IP and TCP headers, for example. In a core network, a packet might begin with a stack of various Carrier Ethernet or MPLS headers, reflecting en-route encapsulation, for example. The basic parsing problem can be formulated as traversing a stack of headers in order to:

- Extract a key from the stack (e.g., a 16-bit packet type field or a TCP/IP five-tuple); and/or
- Ascertain the position of the data payload (e.g. to enable deeper packet inspection).

The traversal is guided by a parsing algorithm consisting of rules for interpreting different types of header format. Note that, without loss of generality, this approach can be extended to the parsing of packet trailers, if required. The parsing process must also smoothly handle failures of parsing, indicating unsupported packet forms. The results of parsing feed into other network processing components. These can include key lookup engines for packet classification, and regular expression matching engines for deep packet inspection.

This paper presents four main contributions which, taken together, offer a flexible and scalable solution to the problems posed by high performance packet parsing:

- Introducing a simple high-level domain-specific language for directly describing packet header parsing algorithms in an object-oriented style.
- Using the concurrent processing capabilities of modern FPGA devices to enable the creation of tailored virtual processing architectures that match the individual needs of particular packet parsing algorithms. These provide the required packet processing performance over a wide range.
- Demonstrating a fast compiler that maps a parsing algorithm description to a matching FPGA-based virtual architecture. This removes existing barriers to ease of use of FPGAs by hardware non-experts, and also facilitates experimentation with the operational characteristics of the implementation.
- Embodying programmability into the virtual architecture, so that the parsing algorithm can be updated dynamically during system operation.

978-0-7695-4521-9/11 $26.00 © 2011 IEEE
DOI 10.1109/ANCS.2011.12

IEEE computer society

1.1 The Packet Parsing (PP) Language

PP was specifically designed to provide a high-level way of describing formats of packet headers and rules for parsing these headers. Thus it is a very simple domain-specific language, not a general-purpose programming language. PP is completely protocol-agnostic, with no built-in restrictions on packet formats. The aim is that the PP user can concentrate on packets, and automatically obtain high-performance results. PP does not involve specifying any details of the machinery used for parsing rule application.

Conceptually, the PP parsing rules can be seen as embedded within the following standard 'outer loops':

```
while true do {
   input packet;
   header := first header;
   while not done do {
      apply rules for header;
      header := next header;
   }
   output packet and results;
}
```

Thus, the PP description omits the standard control flow descriptions that would be needed in a typical programming language description. The lack of user-specified administrative detail in PP makes it particularly appropriate for efficient implementation on target technologies that support parallel execution with streaming data flows.

Section 2 of the paper introduces the PP language.

1.2 FPGA Technology

Traditional approaches to providing the required flexibility in packet parsing involve using general purpose servers as a basis for network nodes. However, these may not be capable of providing the required performance. To address this, the combination of general purpose processors and specialized high-performance network processors is possible. However, the increasing specialization of network processors can thwart goals of flexibility and scalability. The Field Programmable Gate Array (FPGA) is an alternative technology that can fulfill the necessary requirements for high-speed concurrent packet processing, and which can be harnessed in tandem with complementary general-purpose processors.

A simplistic view of an FPGA is that it just comprises a two-dimensional array of programmable logic gates, together with programmable interconnection of logic gates to form logic circuitry. However, the modern FPGA device is a very complex system on chip, including also memory blocks, multiplier-accumulator units, and embedded processors, for example. Thus, the FPGA is now a parallel assemblage of diverse programmable components, with a programmable interconnection network between these components. The big challenge though is to make this raw computational substrate available for easy use by the networking system designer.

The large, and increasing, sizes of the programmable logic array alone (for example, the largest devices now have over 2,000,000 programmable logic cells) mean that circuit designs can be very complex. Added to this is the further complexity of targeting the overall programmable system on chip capabilities. Almost no-one now tackles implementation at the level of the basic FPGA capabilities. The standard approach, reflecting hardware design in general, is to use hardware description languages, such as Verilog and VHDL, as a starting point. While this is distinctly higher level than raw logic design, it is still hard and requires expert empathy with the technology, for example, through attention to timing detail and signaling detail. This hardware haze obscures a higher-level view of the functions that are being implemented.

A key feature of using PP is the much higher level of description language, supported by its efficient compilation to FPGA-based implementations. The effect is to unveil the capabilities of the FPGA to the networking expert. Section 3 of the paper describes a compiler that processes PP descriptions and generates high-performance FPGA-based implementations that exploit the programmability of the technology. These implementations include extremely wide (up to 2048-bit) parallel data paths for streaming packet data through heavily pipelined tailored function units. Section 4 of the paper describes the pipeline stage micro-architecture, and how it can be reprogrammed while in operation, to allow dynamic modifications and upgrades reflected in changes to the original PP description.

1.3 Benchmarking

A benchmark suite of 10 examples described using PP was constructed, based on real requirements reported by a number of major telecommunication equipment vendors. These include examples that are predominantly 'layer 2', including MPLS labels and Ethernet and VLAN headers, and also predominantly 'layer 3 and above', including IPv4, IPv6, TCP, UDP and RTP headers. Section 5 of the paper introduces the suite, and includes a detailed explanation of the PP compilation process for one example.

Experimental results are reported in Section 6. These are all targeted at the Xilinx Virtex-7 HT FPGA device. These results illustrate the scalability of the PP approach by showing that a wide range of packet throughputs can be obtained through varying the instructions given to the compiler. In particular, the results show that 400 Gb/s throughput can be obtained (although, in fact, the parser is capable of a raw 600 Gb/s throughput). The results also indicate scaling of the amount of FPGA resource required and the parsing latency, for the different throughputs.

1.4 Remainder of Paper

Section 7 contains a discussion of related work, and the paper closes with a summary of conclusions and future directions in Section 8.

2. PP Language

The Packet Parsing (PP) language treats packets in an object-oriented style, in order to provide a familiar model for software engineers. In a PP description, an object class is defined for each kind of packet header that is to be parsed. Then the header stack of a packet is conceptually viewed as being a linked list of objects, one per header, the class of each object corresponding to the header type. The definition of a class contains two parts:

- A *structure* which defines the format of the header in terms of an ordered list of fields; and
- A set of five standard *methods* (three optional) which define parsing rules for this header type.

The syntax of PP was minimized so that the user needs only write down what is required for packet parsing. As an experiment, a sugared form of the language was strictly aligned with the Java syntax for class declarations, thereby providing embedding in a standard programming language. However, typical descriptions were doubled in length, due to redundant features and verbose style.

An example class declaration, for IPv4 header parsing, is:

```
class IPv4 {

  struct { version  : 4,
           hdrLen   : 4,
           tos      : 8,
           length   : 16,
           id       : 16,
           flags    : 3,
           offset   : 13,
           ttl      : 8,
           protocol : 8,
           hdrChks  : 16,
           srcAddr  : 32,
           dstAddr  : 32,
           options  : *
  }

  method next_header = protocol;
  method header_size = hdrLen*32;
  method key_builder =
     {srcAddr, dstAddr, protocol};
  method earliest = 2;
  /* method latest = OMITTED */

}
```

Here, the `struct` part lists the well-known IPv4 header fields, with widths in bits, in transmission order. The `options` field has a 'wild card' width, indicating that it is not statically determinable in advance.

The two compulsory methods are `next_header` and `header_size`, which guide the parsing algorithm. The `next_header` method computes the class of the next header to be parsed, as an unsigned integer value (here just the value of the `protocol` field). A special `done()` expression can be used to indicate completion of parsing.

The `header_size` method computes the size in bits of the header being parsed, and thence the offset of the next header within the packet. Here it is just the value of the `hdrLen` field (IPv4 header length in 32-bit words) multiplied by 32. A special `size()` expression can be used to give the size of the header if it can be statically determined in advance. It is easy to see how, together, these two methods can steer the parsing of a header stack.

The final piece of describing the parsing algorithm is that the PP description must have a starting class and a starting offset. If the former is not specified, then the first class appearing in the PP description is used. If the latter is not specified, then a zero offset is used.

The `key_builder` method is optional. It is used to define a contribution to the parsing result from a header object. This result is accumulated as parsing proceeds. Here, the `srcAddr`, `dstAddr` and `protocol` fields are included. For example, if followed by TCP header parsing that contributes source and destination port numbers, this would provide the basis for generating a standard TCP/IP five-tuple as a result of the packet parsing. If no key building is included, then the default is to provide the final class number (i.e., packet header type) as a parsing result.

The remaining two methods, `earliest` and `latest`, are optional assistance to the PP compiler. These indicate the earliest and latest points, respectively, at which this header type can occur in a header stack. Here, for example, the `earliest` method is giving the value 2, indicating that the IPv4 header will have at least one layer of encapsulation (e.g., within an Ethernet packet). For some PP descriptions, with very statically defined header formats and parsing rules, the PP compiler can work out earliest and latest values for each class. However, in general, header stacks are dynamic on a per-packet basis.

Although the right-hand sides of the method declarations in this example are very simple, PP allows arbitrary expressions, in terms of packet field values and constants, to be used for `next_header`, `header_size`, and `key_builder`. These can be explicitly coerced to required bit widths, if necessary. In particular, conditional if-then-else expressions are allowed in order to express more complex parsing behavior. Instances of this can be seen in the benchmarking example presented in Section 5.

Note that the current form of PP involves read-only access to packets. In the future, it could be extended to include packet modification as a side effect of the parsing process.

As stated in Section 1.1, and as can be seen from the overview provided here, PP descriptions are completely implementation independent, saying nothing about how packet data is presented for parsing, or how the parsing algorithm is implemented. PP is thus suitable for either hardware or software implementation. Any additional implementation information is provided to a PP compiler, rather than being part of the PP description.

3. Compiling PP to Programmable Logic

The main goal for the FPGA-based parsing implementation was to achieve packet throughput in the 100s of Gb/s range, employing a scalable approach that would not require substantial re-engineering with each new step in required throughput. The physical constraints were the amount of programmable logic available on target FPGA devices, and the achievable clock rates for such logic. The typical range for the latter is between 200 and 400 MHz, assuming fairly careful logic design. Because of this, it is necessary to use wide data paths, for example, a 512-bit or 1024-bit data path width to obtain an overall 200 Gb/s data rate.

The setting for the packet parsing module generated by the PP compiler is one involving the streaming of packet data through the module, using a very wide data path. In some cases, this packet data might just consist of the relevant header part, following payload offload to temporary memory; in other cases, notably initial packet classification, this data is the entire packet. The packet parsing is performed on the fly as the packets stream through. In other words, the module has cut-through operation, rather than store and forward, which introduces higher packet processing latency.

In order to achieve clock frequencies in the desired range, pipelining is deployed extensively in the packet parsing module generated by the compiler.

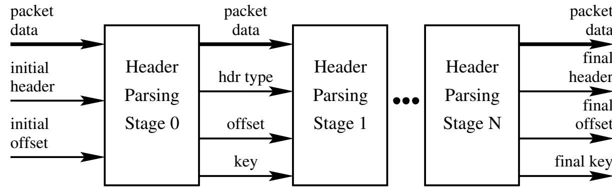

Figure 1. Packet parsing pipeline architecture

Figure 1 shows the top-level architecture. The two basic dimensions are the width of the packet data path (which determines the raw throughput), and the length of the parsing pipeline (which depends on the complexity of the parsing algorithm). The PP compiler performs a natural mapping between the parsing algorithm and the pipeline: there is one pipeline stage for each level in a packet header stack. That is, as a packet advances through the pipeline, one header is parsed at each stage. In steady state operation, multiple packets are being parsed simultaneously in the pipeline.

To guide the basic architectural dimensioning, two parameters are supplied to the PP compiler by its user. The first is a target throughput: this leads to selection of a data bus width based on expected clock frequency. The second is a maximum parse depth (i.e., maximum header stack size): this leads to selection of the number of pipeline stages. Based on an analysis of the PP description, the compiler determines which subset of the defined header types could occur at each of the pipeline stages. User specification of `earliest` and/or `latest` methods within classes assists in this determination.

After this analysis, the compiler generates pipeline stage implementations that are customized to handle precisely the right subset of headers. When a packet enters a stage, it is accompanied by two critical pieces of control information. The first identifies the header type to be parsed at that stage, and the second identifies the offset within the packet where parsing should start. In terms of the PP description, these two values correspond to the `next_header` and `header_size` method results from the previous stage.

Note that the pipeline width and length, and the contents of each pipeline stage, are fully customized for the particular parsing algorithm given in the PP description. Creation of a bespoke virtual architecture is a significant feature of using programmable logic as the implementation medium. This is in contrast to using ASSP, CPU, or NPU, technologies, where the non-trivial task is to map problem instances onto fixed architectures efficiently. Here, the architecture is mapped on to the problem instance. The results of this work, as reported in Section 6, indicate that the indirection through programmable logic does not impede the achievement of required performance. Further, the precise architectural customization might reduce power consumption, through omitting redundant components that are present in 'one size fits all' fixed architectures.

The basic function of each pipeline stage is to evaluate the expressions for the `next_header`, `header_size`, and (if present) `key_builder`, methods that feature in the class for the header type selected for parsing of the packet that is passing through this stage. This involves obtaining the values of all the packet header fields that feature in these expressions, by extracting them from the wide words of the packet as they are streamed through the stage. Note that individual fields may overlap one or more words, depending on the exact packet structure and its mapping onto the selected data path width. Multiple packets may also overlap in the same word, especially with very wide data buses. For example, a 64-byte minimum-size Ethernet packet fits within a single 512-bit data path region.

Each pipeline stage generated by the compiler contains customized logic to perform packet field extraction. This involves counting until the first word of interest, and then shifting and masking to form each field value. Note that these tasks are non-trivial given the data path widths and the desired clock rates, and so careful logic design was necessary. The stage also contains arithmetic logic to compute the expressions. This is heavily pipelined, in order to maintain the necessary clock rate. The extent of the pipelining depends on the complexity of the expressions. In practice (as will be seen in Section 5), expressions tend to be relatively simple.

The internal micro-architecture of a stage follows a standard template, connecting five basic components that incorporate the necessary customizations for that stage. Using this template provides timing guarantees for the enclosed logic circuitry and, in addition, supports programmability of pipeline stages during operation.

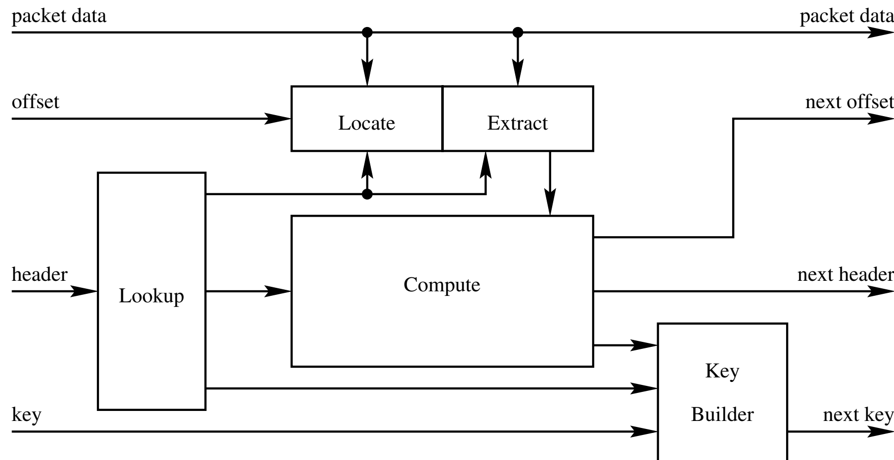

Figure 2. Micro-architecture of a parsing pipeline stage.

4. Parsing Pipeline Stage Micro-architecture

Figure 2 shows the internal micro-architecture template for the pipeline stages generated by the PP compiler. As indicated in Section 3, packet data enters the stage wordwise, and leaves the stage unchanged. A header type value and a packet data offset enter the stage, and new values for both, computed using the methods defined for this header type, leave the stage. Finally, an accumulated key value enters the stage, and an updated key value as specified by the `key_builder` method, leaves the stage.

Functionally, a header parsing stage operates as follows. When a packet starts to arrive at the input of a header parsing stage, it comes in tandem with the header type identifier, the offset in the data stream, and a key being constructed. A header type *lookup* component uses the input header type identifier to fetch customized microcode that programs the remaining components in the stage to be able to handle the particular header type. Meanwhile, the input header offset within the packet stream is forwarded to a *locate* component that finds the header within the input packet stream. The locate component works in tandem with an *extract* component that discovers header fields for use in parsing computations, and key building. A *compute* component performs operations associated with the methods in the parsing description, such as computing the next header and the header size. Results of the compute component can also be forwarded to an optional *key builder* component that constructs a revised parsing key. Each of the five components is described in more detail in the sections that follow.

One reason for using per-packet microcoding is to allow sharing of a common set of components in order to parse any of the header types that must be handled at a particular pipeline stage. This was chosen as a more resource-efficient alternative to having *n* sets of components, one set for each possible header type.

4.1 Lookup Component

The header type lookup component fetches microcode to then program the rest of the stage to handle the particular header type to be parsed. The other main reason for using microcode in each parsing stage is to allow the packet parser to be modified while operating: to add, remove, and/or modify the particular header types that can be parsed in a stage. This aspect is discussed further in Section 4.7.

Microcode stores header offsets and sizes of fields to be extracted as part of computing the header object methods. The microcode also stores information on what compute operations to perform and with what data. Data can be sourced from packet fields or from constants embedded within the microcode. Finally, the microcode additionally indicates whether to update the key result that is accumulated from stage to stage.

Microcode is stored locally in a parsing stage for each possible header that can be parsed within that given stage. Microcode cannot be shared across stages in general due to the fact that different sets of header types may be found in different parsing stages. Different collections of header types result in different configurations of components within the parsing stage, which then each have unique microprogramming requirements.

Depending on how many microcode entries exist in the header parsing stage and the size of each microcode entry, the lookup component will store and retrieve the microcode differently. When relatively few entries are required in a stage, microcode is simply stored locally in flip-flops. Retrieval, then, is merely a matter of implementing a multiplexer. When the number of header types parseable in the stage climbs, the associated microcode entries are more efficiently stored in block memory that is distributed throughout FPGA devices. Retrieval from block memory amounts to simply providing an address, such as the unique identifier given to each header type.

4.2 Locate Component

Packet data streams through each parsing stage in a word-wise fashion. It is the responsibility of the locate component to locate and deserialize the portion of a packet header that contains the fields required for computations within the stage. The input header offset indicates where the header starts in the packet stream. Microcode then provides the starting point within the header, and the size of the region containing the fields of interest. The locate component unpacks the header from the packet stream by handling data word boundary and alignment issues. Since the word width of the streaming packet interface is a parameter to the PP compiler, the template for the locate component is configurable to support different data widths.

Location of the header amounts to counting input words based on the starting point, and then accumulating the contents of as many words as are required to capture the fields of interest. As an example, in the IPv4 parsing description shown in Section 2, the first required field is `hdrLen` and the last required field is `dstAddr`, and so a 156-bit section of the packet would be captured.

4.3 Extract Component

The extract component consists of a number of extraction units, one for each packet field that is required by computations within the stage. Given that the stage must support a set of header types, the number of units required is determined by the header type that uses the largest number of fields. The input to this component is the deserialized packet header segment produced by the locate component. Microcode instructs each extraction unit on the offset and size of its field within this segment.

The extraction is performed by a shifting and masking approach. The implementation of this is non-trivial though, given the need to cater for arbitrarily large offsets and field widths, and to maintain the desired clock rate. A pipelined shifting approach was used, involving 16 choices of shift distance at each stage, the granularity of distance increasing by 16x at each stage. For example, a 92-bit shift to extract the IPv4 `srcAddr` field from a 156-bit segment would involve two successive shifts, by 12 and 80 respectively.

4.4 Compute Component

The compute component consists of a number of compute units, one for each expression that is evaluated in the stage. There are at least two units, one used for `next_header` method computation, the other used for `header_size` method computation, plus an additional number of units (possibly zero) required by the header type that has the largest `key_builder` tuple size. Microcode instructs each unit on the expression to be evaluated, including the sources of its operands and its operators. The PP compiler optimizes each compute unit, and its microcode, so that it has precisely the minimal data path width and functional capabilities that are required to compute a particular method's expression for all of the different header types that are supported at the pipeline stage.

Each compute unit is organized using pipelines for expression evaluation, with a single two-input arithmetic or logical operation being done at each stage. The simple stages allow the desired clock frequency to be maintained. The pipeline stages are organized to carry out a stack-based expression evaluation scheme. An operand/result value stack is passed along the pipeline. On entry, this contains all of the operand values for the expression and, on exit, this contains the result of evaluating the expression. At each stage, the top two values are popped from the stack, combined with a two-input operation, and the result is pushed onto the stack. Microcode selects the operator to be used at each stage, drawn from a range including addition, subtraction, shifting, comparisons, and bit-wise operations.

When the definition of a method includes an if-then-else construct, evaluation of the result is done using three of the above pipelines in parallel. One computes the if-condition, one computes the then-result, and the other computes the else-result. Note that both of these results are computed speculatively to reduce overall delays. The choice between them to determine the final result is made by selection using the if-condition value.

4.5 Key Builder Component

Key building is an optional cumulative process along the parsing pipeline. At each stage, if a header type that features a `key_builder` method is being parsed, then a set of expression results is appended to the key received from the previous stage. The final result from parsing is then a tuple of all the accumulated results. The key is passed between stages as a single parallel word. Microcode instructs the key builder on the number of values to be appended, and the sources of the values. The latter sources are the outputs of the appropriate compute units within the compute component. In many practical cases, as seen in the IPv4 example of Section 2, the values are just packet field values, in which case no computation is required.

4.6 Error and Exception Handling

Parsing exceptions are expected to occur when handling live network traffic. Malformed packets or packets with unrecognized headers are possible. Errors and exceptions do not hinder the operation of the packet parser. In such cases, the packet parser flags that an unparseable packet has been encountered, but packets that are flagged as suspect are still passed out of the pipeline along with an indication of where the error/exception condition occurred. This takes the form of the header type and offset value at the time of the failure. The expectation is that some downstream module will make a decision on the fate of these packets. Header parsing stages transition to a pass-through mode when they see that an incoming packet already has an error/exception flag. In this way, throughput is maintained and packets remain in exact input order.

As a possible future feature, it would not be hard to add support for a compiler option calling for unparseable packets to be dropped within the parser module.

4.7 Programmability

As seen in Sections 4.1 to 4.5, microcode instructions are used extensively to control the behavior of the five components within each parsing pipeline stage. This allows the same set of resources to be shared for each of the different header types being processed by a stage. The exact microcode format is specific to the set of components contained in a particular stage. The size of the stored microcode depends on the complexity of the components.

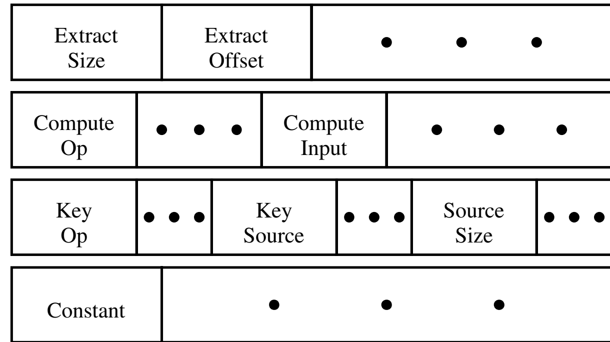

Figure 3. Pipeline stage microcode organization

The general format of the microcode is shown in Figure 3. It consists of four sections. The first section consists of zero or more extract size-offset pairs. These correspond to different fields that may be extracted from the packet in order to parse a header. The size indicates the bit width of the field, and the offset indicates its bit position from the start of the header segment. The second section consists of compute operations and input selectors. One compute operation entry exists for each stage in a compute unit pipeline. The supported operations are encoded as unique integer identifiers. The compute input selectors program a multiplexer to enable the appropriate inputs to reach a compute unit. Multiplexer inputs could be the different extracted fields or constants from the microcode. The third section consists of zero or more sources for data to be appended to the packet's context key. The final section consists of constants, occurring in the header object description and then used directly in computations. Constants can be of variable size.

The overall PP programming approach involves generating customized parsing pipelines from a PP description. The use of microcode enables later parsing algorithm changes without necessarily generating a new microarchitecture. For example, modifications to the method definitions for existing header types may be accommodated unless they involve significantly more complex expressions than the existing extract or compute components can support. Also, addition of new header types, with methods of similar complexity to existing methods, is possible. Clearly, removal of particular header types, or simplification of methods, is easy to accommodate. It is possible for the PP compiler to over-provision a generated microarchitecture, thus leaving some spare capacity for later parsing updates.

5. Benchmark Suite

A main motivation for this research came from telecommunication equipment providers interested in the use of FPGA technology to provide high-performance, yet highly-programmable, packet parsing. The requirement was to provide an open-ended, high-level way to describe packet parsing algorithms, and to allow in-operation updates without FPGA circuitry changes. PP was the resulting solution to these needs. The benchmark suite was drawn from examples required in practical networking situations. These fall into two broad categories: carrier (wide area and metro area networks), and end system (access and enterprise settings). In turn, these categories correspond to layer-two and below, and layer-three and above, protocol settings respectively.

The following 10 examples were included:

- **JustEth**: searches Ethernet frames for the type field – the baseline example.
- **VlanAndMpls**: handles Ethernet with VLAN and MPLS encapsulation.
- **AllStack**: handles a combination of VLAN and MPLS stacking, and continues through the header stack into IPv4 or IPv6, then TCP or UDP.
- **ArpIcmp**: handles ARP and ICMP over Ethernet.
- **TcpIp4**: handles TCP within IP version 4.
- **TcpIp6**: handles TCP within IP version 6.
- **TcpIp4andIp6**: handles TCP within IP version 4 or IP version 6.
- **RtpIp4**: handles RTP protocol within UDP within IP version 4.
- **RtpIp6**: handles RTP protocol within UDP within IP version 6.
- **RtpIp4andIp6**: handles RTP protocol within UDP within IP version 4 or IP version 6.

The parsing result varies between examples. In the first two, the result is a 16-bit standard Ethernet type field and offset for the next encapsulated header. In the next two examples, and the final three examples, the result is the unearthing of the data payload offset. In the remaining three examples, the result is a TCP/IP five-tuple.

Figure 4 shows the complete PP description for the second example. It is drawn from a Carrier Ethernet setting, where an MPLS frame, with some number of MPLS tags, is used to carry an Ethernet frame that can optionally contain some number of VLAN (Virtual LAN) headers. The goal is to reveal the packet being carried within the overall encapsulation: its type, and its offset within the overall packet. This information can then be used for packet classification based on content.

The description contains three classes: for MPLS, Ethernet, and VLAN. The `#define` statements at the beginning make a simplifying connection between the internal values used to identify header types and actual values used by the standard IEEE 16-bit type numbering scheme.

Within each class, the `struct` part shows the familiar formats for each of these packet header types. The methods then express the parsing algorithm. Note that the parse will start by default with the MPLS class since it appears first. In all three classes, the `next_offset` method just returns the `size()` value, which is the length of the current header (32, 112, or 32 bits respectively). Also, these classes contain no `key_builder` methods, meaning that the parsing result will just be the final next header type value from the parsing chain.

The `next_header` method in the `MPLS_TYPE` class expresses the fact that there will be some number of MPLS tags with the `S` (bottom of stack) bit equal to zero, followed by the final tag with the `S` bit set to one. At this point, an Ethernet frame will be expected. Note that the MPLS tag format does not include an explicit indication of the type of the MPLS payload.

Then, the `next_header` method in the `ETH_TYPE` class just takes the type field from the Ethernet header in order to determine the next header type to be parsed.

```
#define MPLS_TYPE       0x8847
#define ETH_TYPE        0x0001
#define VLAN_TYPE       0x8100

class MPLS_TYPE {
  struct {
    label : 20,
    cos   : 3,
    sBit  : 1,
    ttl   : 8
  }
  method next_header =
    if (sBit == 0)
      MPLS_TYPE;
    else
      ETH_TYPE;
  method next_offset = size();
}

class ETH_TYPE {
  struct {
    dmac : 48,
    smac : 48,
    type : 16
  }
  method next_header = type;
  method next_offset = size();
}

class VLAN_TYPE {
  struct {
    pcp : 3,
    cfi : 1,
    vid : 12,
    tpid : 16
  }
  method next_header =
    if (tpid == VLAN_TYPE)
      tpid;
    else
      done(tpid);
  method next_offset = size();
}
```

Figure 4. PP source code for VlanAndMpls example

At this point, the parse will fail unless this value is equal to one of the three values that have corresponding classes in the PP description. For more safety, an alternative here would be to check explicitly that `type` is equal to `VLAN_type`.

Finally, the `next_header` method in the `VLAN_TYPE` class expresses the fact that there will be some number of VLAN tags with their `tpid` field indicating a further VLAN tag within, followed by the final tag with a `tpid` field indicating something different. At this point, the special `done()` function is used to indicate that parsing is complete, its argument being the `next_header` result.

The PP compiler can infer from the description that the MPLS, Ethernet, and VLAN, classes can occur earliest at the first, second, and third places in the header stack respectively. It cannot infer anything about their latest positions in the stack. Thus, the generated parsing pipeline has appropriate provisioning

Table 1. Benchmark results for 1024-bit data path

	Resource utilization (% FPGA)	Clock period (ns)	Raw T'put (Gb/s)	Total latency (ns)
JustEth	9.2	2.985	343	293
VlanAndMpls	11.0	2.999	341	309
AllStack	11.5	3.389	302	349
ArpICMP	14.9	3.345	306	495
TcpIp4	12.1	2.984	343	292
TcpIp6	9.9	2.987	343	293
TcpIP4andIP6	12.4	3.154	325	309
RtpIp4	12.6	3.078	333	348
RtpIp6	10.8	3.133	327	354
RtpIp4andIp6	13.0	3.131	327	354

Table 2. Benchmark results for 2048-bit data path

	Resource utilization (% FPGA)	Clock period (ns)	Raw T'put (Gb/s)	Total latency (ns)
JustEth	17.2	2.983	687	292
VlanAndMpls	18.9	3.068	668	316
AllStack	22.7	3.542	578	365
ArpICMP	23.1	3.648	561	540
TcpIp4	20.9	3.117	657	305
TcpIp6	17.7	3.338	614	327
TcpIP4andIP6	21.0	3.243	632	318
RtpIp4	22.2	3.468	591	392
RtpIp6	19.0	2.992	685	338
RtpIp4andIp6	22.6	3.205	639	362

for the possible sets of header types in its stages. In particular, all stages from the third onward have provisioning for all three header types.

In this example, the expressions occurring in the methods for each class only require one packet field each time, and so the generated extract component at each stage has only one extract unit, which has an extractee width of 16 bits. Since the method expressions are simple, the compute component only requires one compute unit that can perform an equality comparison operation, plus one if-then-else parallel compute unit. These all have widths of 16 bits, since all operands and results have this width. The three respective `size()` 'function calls' just involve the insertion of compile-time constants into the compute component microcode.

The microcode word size for the three-header type stage was 136 bits, giving a total storage requirement of 408 bits at each stage, which can easily be provided by local storage in flip-flops. In fact the range of microcode word sizes over the benchmark suite was from 40 bits (JustEth) to 372 bits (ArpIcmp), the average being 140 bits.

To illustrate possible re-programmability without the need to change the generated pipeline architecture, it can be seen that the above suggestion of including a conditional test on the type value during Ethernet header parsing could be added by microcode update, given the availability of a spare if-then-else capability in the compute component.

6. Experimental Results

The examples in the benchmark suite were implemented for the Xilinx Virtex-7 870HT FPGA. This FPGA was chosen because it includes 16 28 Gb/s and 72 13.1 Gb/s serial transceivers. Thus, one future setting for the parser would be with packet input via a 400 Gb/s Ethernet MAC attached to 16 28 Gb/s transceivers, and packet output via a 600 Gb/s Interlaken interface attached to 48 12.5 Gb/s transceivers. The FPGA contains 136,900 'slices', each containing four six-input lookup tables (LUTs) and eight flip-flops (FFs).

The PP compiler generates a description of the customized pipeline parsing architecture in either VHDL or Verilog hardware description language, VHDL being chosen here. These descriptions were then processed by the standard Xilinx ISE 13.1 design tool suite, which performed synthesis, placement, routing, and bitstream (FPGA physical programming information) generation. A feature of the PP design environment is that, when the complete FPGA implementation process has been done once, it indicates whether subsequent modifications to the parsing algorithm can be implemented just by making microcode updates, or whether a revised architecture has to be generated. The PP environment includes a software driver that is used to write to microcode memory on the FPGA. The correctness of the FPGA implementations was validated using simulation of the (pending) Virtex-7 FPGA and a surrounding 400 Gb/s networking setting.

For all the experiments, a maximum parsing depth of five headers was specified to the PP compiler, this being enough to be realistic for all of the examples. A range of target throughputs was requested, to explore the scalability of the approach. These translated into different data path widths in the parsing pipeline architecture. The FPGA resource utilizations, and achieved performance, were measured.

Tables 1 and 2 contain experimental results for 1024-bit and 2048-bit data paths, respectively. The first column shows FPGA resource utilization, in terms of the percentage of the Virtex-7 870HT slices used for the implementation. Estimates for the combined sizes of future 400 Gb/s Ethernet MAC and 600 Gb/s Interlaken interface blocks are approximately 65% of this FPGA, and so a combined bridging and parsing subsystem contained on a single Virtex-7 family FPGA is practicable.

The clock periods shown in the second column translate into clock rates of between 274 and 335 MHz, which is in the range that the carefully pipelined architecture was designed to guarantee. These clock rates then translate into the raw throughput shown in the third column, obtained by multiplying by the data path width. It can be seen that the 1024-bit data path gives results just below the 'headline rate' of 400 Gb/s, whereas the 2048-bit data path gives results well above this. However, caution must be taken with these raw throughput figures, as the effect of short packets and quantization over a wide word size must be taken into account.

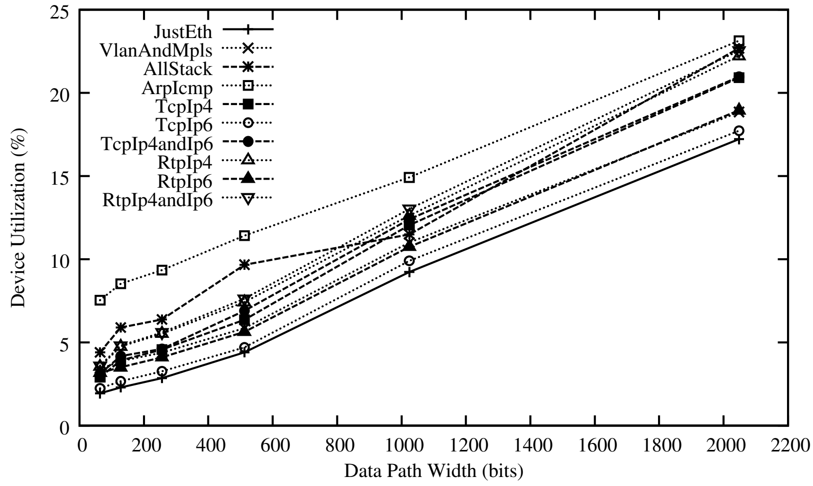

Figure 5. Resource utilization versus data path width

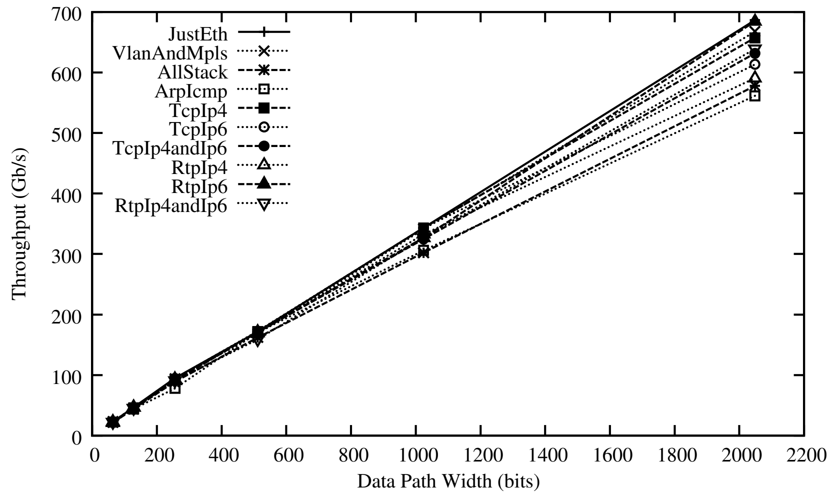

Figure 6. Raw throughput versus data path width

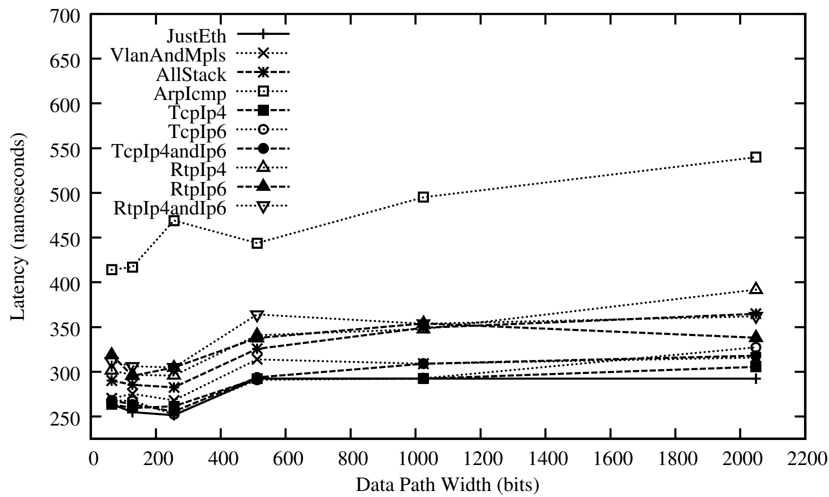

Figure 7. Total latency versus data path width

For example, if only one packet per word is allowed, then a series of minimum-size 64-byte Ethernet packets would only occupy 25% of the 2048-bit data path width, sharply reducing the real throughput.

To guarantee full 400 Gb/s data throughput for minimum-size Ethernet packets, that is, handling 600 million packets per second, one solution is to divide the 2048-bit wide data path into eight 256-bit logical lanes, allowing a new packet to start in alignment with any lane. With this arrangement, a maximum of four packets might be in flight during one word time, which can be handled directly using four parallel copies of the parsing pipeline. For the benchmark examples, this parallel solution, with an Interlaken interface, would suit a Xilinx Virtex-7 1140XT FPGA.

The final column in the tables shows the latency of the packet parsing. Because of the basic pipeline architecture, the latency is essentially directly proportional to the selected parsing depth, which was fixed as five in these experiments. The average latency per header parsed was between 58 and 108 ns in the 2048-bit data path case. This latency represents a trade-off against the need for high throughput, achieved by the use of large-scale pipelining. Note that the latency does not introduce any need for temporary packet buffering elsewhere outside the parsing module, since the packets are stored in a distributed manner as they pass through the parsing pipeline. The latencies reported here are deterministic, and are acceptable for various practical scenarios within future high-throughput telecommunications equipment.

Figures 5, 6, and 7 show experimental results across a wider range of data path widths: 64, 128, 256, 512, 1024, and 2048. These demonstrate the scalability of the PP approach, since the different implementations were all derived by compiling the same PP descriptions. At the low end of the scale, these results show that a 64-bit data path can supply 20 Gb/s packet throughput using relatively modest FPGA resources. It can be seen that resource usage does not double with each doubling of the data path width, indeed increases with a much less steep linear function. This is because the bulk of the resource is consumed by the compute components. Since these operate on narrow operands, extracted from the very wide data path, they remain the same size with upwards scaling.

The throughput increases in step with the data path width increases, as intended. This is a little less than doubling with each width doubling and shows increasing variability between examples, because of more challenging layout of wide data paths on the FPGA that results in some reduction in the achievable clock rate. The latency remains largely flat with increasing data path width, essentially because the extent of pipelining remains the same for all data path widths above the 512-bit data width threshold.

A further experiment used a data path width of 4096, which gave results consistent with scaling upwards further. However, as discussed earlier, the multiple packet per word problem is more acute here, and requires further mechanisms. This is a matter that will be addressed in future work.

7. Related Work

There are three main areas of prior work relating to this research: technologies for packet processing at rates up to 100 Gb/s; packet lookup and classification; and languages for packet processing and/or targeting programmable logic.

Karras et al. [11] present a folded pipeline architecture for 100 Gb/s carrier networking, handling both MPLS labels and PBB Carrier Ethernet. Wu et al. [28] discuss the simplification of data path processing in next generation routers. Mudigonda et al. [20] discuss the impact of the 'memory wall' on high-speed packet processing. Current specialized commercial devices extend to 40 Gb/s rates, for example the NetLogic knowledge-based processor families [23], or the Cavium multi-core processor plus acceleration families [5]. The latest NetFPGA platform [22] supports FPGA-based networking research at up to 40 Gb/s rates.

An increasingly important topic is the introduction of new protocols into networking equipment. Anwer et al. [1] describe Switchblade, a platform for rapid deployment of network protocols on programmable hardware. Carli et al. [4] describe PLUG, a means for deploying flexible lookup modules in high-speed routers. OpenFlow [21] provides an open framework for enhancing packet routers.

Kobiersky et al. [13] utilize an XML description to auto-generate finite state machines for protocol handling at up to 20 Gb/s rates. The XML enumerates the FSM transitions, one per possible parsing path. Packets are streamed through the system, and each byte is checked for relevant fields to extract. A concern with this work is scalability. The use of a crossbar in the extraction unit will have difficulty scaling with increasing data path widths. Additionally, when a larger number of protocols are to be handled, the generated state machine could become a performance bottleneck.

The Kangaroo packet parsing architecture of Kozanitis et al. [15] can deal with 40 Gb/s line rates. It utilizes a TCAM to enable speculative fetching of pre-defined offsets in a packet. Based on which TCAM entry matches, the packet format is known. The TCAM entry returns the next instruction to perform. Packets are stored in memory, and instructions dictate packet fields to be fetched from the stored packet. The lookahead stride for protocols to handle is, therefore, limited by how many fields must be fetched from the memory subsystem. An offline algorithm is used to construct the TCAM entries.

Most published research on packet classification has focused on two other topics, IP address lookup for routing and matching against rule sets, with initial packet parsing presumed done in advance. Prasanna et al. [10,17,29] have demonstrated IP address lookup at up to 100 Gb/s rates using FPGA implementation; other IP address lookup research includes [16,18,25,27]. Packet matching research is typically based on the Snort [2] rule-based intrusion detection technology. Examples of packet classification that match against rule sets at up to 20 Gb/s rates include [12,19,26].

A number of domain-specific programming languages for packet processing have been proposed, albeit not usually targeted for programmable logic, for example, Baker [6], FPL [7], and PacketC [9]. Click [14] is well-known as a system for building software routers. Rubow et al. [24] show an environment for targeting programmable logic using a Click approach. Brebner [3] introduced the G language for packet inspection and editing, targeted at FPGA implementation at up to 20 Gb/s data rates.

There has been considerable research into high-level synthesis, i.e., mapping general-purpose languages (usually C or C++ based) onto programmable logic. Much of this concerns efficient loop unrolling to form pipelines. For example, Cong and Zou [8] have studied dynamic nested loops. This work has most application in digital signal processing rather than packet processing.

8. Conclusions and future work

Overall, the experimental results confirm that the modern FPGA possesses sufficient programmable computational capabilities to process packets at very high line rates. The research in particular shows that this raw, distributed, and fine-grain power can be harnessed by the networking expert, through the automatic building of customized packet processing architectures, driven by high-level domain-specific programming descriptions.

The larger context for this research is to embed the packet parsing capability within a complete network processing system. The first step (already demonstrated at a 100 Gb/s line rate) is coupling the parsing module with a key lookup module, in order to perform complete packet classification. The latter module was also generated automatically from a high-level description, employing a number of heavily pipelined architecture templates based on the pioneering work of Prasanna et al. In turn, this packet classification subsystem is being coupled with a traffic management subsystem, to demonstrate a complete 400 Gb/s network processor using a dual Xilinx Virtex-7 implementation.

9. REFERENCES

[1] Anwer, M. B., Motiwala, M., bin Tariq, M., and Feamster, N. 2010, Switchblade: a platform for rapid deployment of network protocols on programmable hardware. In Proceedings of the ACM SIGCOMM Conference (New Delhi, Aug. 2010), 183-194.

[2] Baker, A., Esler, J., and Alder, R. 2007. Snort IDS and IPS Toolkit. Syngress Publishing, Inc.

[3] Brebner, G. 2009, Packets everywhere: the great opportunity for field programmable technology. In Proceedings of the IEEE International Conference on Field-Programmable Technology (Sydney, Australia, Dec. 2009), 1-10.

[4] Carli, L. D., Pan, Y., Kumar, A., Estan, C., and Sankaralingam, K. 2209. PLUG: flexible lookup modules for rapid deployment of new protocols in high-speed routers. In Proceedings of the ACM SIGCOMM Conference (Barcelona, Spain, Aug. 2009), 207-218.

[5] Cavium Networks. http://www.caviumnetworks.com.

[6] Chen, M., et al. 2005. Shangri-La: achieving high performance from compiled network applications while enabling ease of programming. Proceedings of the 2005 ACM SIGPLAN Conference (Chicago, IL USA, Jun. 2005), 224-236.

[7] Comer, D. 2004. Network System Design Using Network Processors, Agere version, Prentice-Hall, Inc.

[8] Cong, J., and Zou, Y. 2010. A comparative study on the architecture templates for dynamic nested loops. In Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (Charlotte, NC USA, May 2010), 251-254.

[9] Duncan, R., and Jungck, P. 2009. PacketC language for high performance packet processing. Proceedings of the 11th IEEE International Conference on High Performance Computing and Communications (Seoul, Korea, Jun. 2009), 450-457.

[10] Jiang, W., and Prasanna, V. 2007. A memory-balanced linear pipeline architecture for trie-based IP lookup. In Proceedings of the 15th Annual IEEE Symposium on High-Performance Interconnects (Stanford, CA USA, Aug. 2007), 83-90.

[11] Karras, K., Wild, T., and Herkersdorf, A. 2010. A folded pipeline network processor architecture for 100 Gbit/s networks. In Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (La Jolla, CA USA, Oct. 2010), 2:1-2:11.

[12] Kennedy, A., Wang, X., Liu, Z., and Liu, B. 2008. Low power architecture for high speed packet classification. In Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (San Jose, CA USA, Nov. 2008), 131-140.

[13] Kobierský, P., Kořenek, J., and Polčák, L. 2009. Packet header analysis and field extraction for multigigabit networks. In Proceedings of the IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems (Liberec, Czech Republic, Apr. 2009), 96–101.

[14] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. 2000. The Click modular router. ACM Transactions on Computer Systems 18 (Aug. 2000), 263-297.

[15] Kozanitis, C., Huber, J., Singh, S., and Varghese, G. 2010. Leaping multiple headers in a single bound: wire-speed parsing using the Kangaroo system. In Proceedings of the 29th IEEE Conference on Computer Communications (San Diego, CA USA, Mar. 2010), 830–838.

[16] Lakshminarayanan, K., Rangarajan, A., and Venkatachary, S. 2005. Algorithms for advanced packet classification with ternary CAMs. In Proceedings of the ACM SIGCOMM Conference (Philadelphia, PA USA, Aug. 2005), 193-204.

[17] Le, H., Jiang, W., and Prasanna, V. 2008. Scalable high-throughput SRAM-based architecture for IP-lookup using FPGA. In Proceedings of 18th International Conference on Field Programmable Logic and Applications (Heidelberg, Germany, Sept. 2008), 137-142.

[18] Lim, H., and Mun, J. 2007. High-speed packet classification using 2-dimensional binary search on length. In Proceedings of the 3rd ACM/IEEE Symposium on Architectures for Networking and Communications Systems (Orlando, FL USA, Dec. 2007), 137-144.

[19] Mitra, A., Najjr, W., and Bhuyan, L. 2007. Compiling PCRE to FPGA for accelerating SNORT IDS. Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems (Orlando, FL USA, Aug. 2007), 127-136.

[20] Mudigonda, J., Vin, H., and Yavatkar, R. 2005. Overcoming the memory wall in packet processing: Hammers or ladders? In Proceedings of the 1st ACM/IEEE Symposium on Architectures for Networking and Communications Systems (Princeton, NJ USA, Oct. 2005), 1-10.

[21] Naous, J., Erickson, D., Covington, A., Appenzeller, G., and McKeown, N. 2008. Implementing and deploying an OpenFlow switch on the NetFPGA platform. In Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (San Jose, CA USA, Nov. 2008), 1-9.

[22] NetFPGA. http://www.netfpga.org.

[23] NetLogic Microsystems. http://www.netlogicmicro.com.

[24] Rubow, E., McGeer, R., Mogul, J., and Vahdat, A. 2010. Chimpp: A Click-based programming and simulation environment for reconfigurable networking hardware. In Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (La Jolla, CA USA, Oct. 2010), 36:1-36:10.

[25] Sourdis, I., Stefanakis, G., de Smet, R., and Gaydadjiev, G. 2009. Range tries for scalable address lookup. In Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, (Princeton, NJ USA, Oct. 2009), 143-152.

[26] Tao, Z., Yonggang, W., Lijun, Z., and Yang, Y. 2009. High throughput architecture for packet classification using FPGA. In Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (Princeton, NJ USA, Oct. 2009), 62-63 .

[27] Vamanan, B., Voskuilen, G., and Vijaykumar, T. 2010. Efficuts: optimizing packet classification for memory and throughput. In Proceedings of the ACM SIGCOMM Conference (New Delhi, India, Aug. 2010), 207-218.

[28] Wu, Q., Chasaki, D., and Wolf, T. 2009. Simplifying data path processing in next-generation routers. In Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (Princeton, NJ USA, Oct. 2009), 11-19.

[29] Yang, Y., and Prasanna, V. 2010. High Throughput and Large Capacity Pipelined Dynamic Search Tree on FPGA. In Proceedings of the 18th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (Monterey, CA USA, Feb. 2010), 83-92.