Ruler: High-Speed Packet Matching and Rewriting on NPUs

Tomas Hruby
Vrije Universiteit Amsterdam
World45 Ltd.
thruby@world45.com

Kees van Reeuwijk
Vrije Universiteit Amsterdam
reeuwijk@cs.vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

ABSTRACT

Programming specialized network processors (NPU) is inherently difficult. Unlike mainstream processors where architectural features such as out-of-order execution and caches hide most of the complexities of efficient program execution, programmers of NPUs face a 'bare-metal' view of the architecture. They have to deal with a multithreaded environment with a high degree of parallelism, pipelining and multiple, heterogeneous, execution units and memory banks. Software development on such architectures is expensive. Moreover, different NPUs, even within the same family, differ considerably in their architecture, making portability of the software a major concern. At the same time expensive network processing applications based on deep packet inspection are both increasingly important and increasingly difficult to realize due to high link rates. They could potentially benefit greatly from the hardware features offered by NPUs, provided they were easy to use. We therefore propose to use more abstract programming models that hide much of the complexity of 'bare-metal' architectures from the programmer. In this paper, we present one such programming model: Ruler, a flexible high-level language for deep packet inspection (DPI) and packet rewriting that is easy to learn, platform independent and lets the programmer concentrate on the functionality of the application. Ruler provides packet matching and rewriting based on regular expressions. We describe our implementation on the Intel IXP2xxx NPU and show how it provides versatile packet processing at gigabit line rates.

Categories and Subject Descriptors

C.2.0 [**Computer-communication networks**]: General—*Security and protection (e.g., firewalls)*; C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems

General Terms

Design, Performance, Security

Keywords

TDFA, regular expressions, deep packet inspection, network processors

1. INTRODUCTION

Programming specialized network processors (NPUs) is an inherently difficult task. Since the role of the NPU is to deliver high performance network traffic processing at multi-gigabit rate, processor cycles are scarce and memory latency is high, so optimization is crucial. As a result, the architectures generally do not offer the convenient abstractions that mainstream processors offer: programmers face a multithreaded environment with high degrees of parallelism, pipelining and heterogeneity of multiple execution units. Programming such a processor is therefore very different from traditional approaches. Moreover, highly optimized solutions are usually complex, poorly portable, and difficult to maintain.

At the same time, there is an increasing need to perform intensive network processing. Deep packet inspection, classification and rewriting are crucial in many application domains. Examples are intrusion detection, data reduction, QoS provision, HTTP load balancing, traffic engineering, anonymization, NAT, network debugging, and filtering. Unfortunately, the growth of link rates easily matches (and perhaps even exceeds) Moore's law [12], and far exceeds the rate at which memory latency improves. As a result, such intensive processing of network traffic is increasingly difficult. In our opinion, architectures like NPUs offer the appropriate hardware support for handling high link rates (e.g., asynchronous memory access), but programming them is (too) difficult.

The language of choice for most system programmers is C, therefore many vendors have tried to support C on their NPUs. Despite a lot of effort to ease such programming [5, 22], generating highly optimized code for different architectures and applications is not trivial and hand-written assembler often outperforms compilers' results. One problem is that the expressiveness of C is limited and does not cover all the arcane features of NPUs. Each manufacturer creates its own C dialect so programmers can deal with the hardware. Porting such code to another platform is very difficult. Characteristics of different NPUs vary a lot, hence programmers must use different coding patterns.

In this paper, we show that it is possible to make programming of NPUs both simple and efficient for the specific application domain of pattern matching and packet rewriting. For this purpose, we discuss Ruler, a domain-specific, rule-based language specifically designed for this application area. Ruler hides the complexity of the underlying hardware and makes programming of NPUs more widely available. It is a language tailored to a specific range of applications and enables the user to focus on quality and correctness of the application without spending time on arcane performance issues. All responsibility for generating efficient code is pushed to the Ruler compiler.

Most existing NPU software for deep packet inspection limits itself to matching byte patterns either using fast matching algorithms such as (improved) Aho-Corasick [2,3], or hardware assists such as CAMs [24]. By implementing regular expressions we go beyond such simple methods, allowing much more complex applications.

Regular expression matching on network devices is currently a hot research topic [16, 17] and we are not the first to offer high-speed regular expression *matching* of network packets. Existing matching engines typically use deterministic finite automata (DFA), as they are both fast and easy to implement [23]. However, in Ruler we also *rewrite* the matched packets to a new form. This introduces a significant complication, because traditional DFAs do not record intermediate positions in the match; they only determine whether an input matches or not. In contrast, we use what is known as *tagged* DFAs [19], where these intermediate positions can be recorded.

In summary, to the best of our knowledge we are the first to apply tagged DFAs to the field of high-speed network processing and to implement a regular expression *matcher* and *rewriter* in this domain. Moreover, Ruler is designed for portability, speed, and ease of programming. Our IXP2xxx implementation is suitable for deployment on the network edge to monitor, classify, filter and rewrite traffic on gigabit networks. In addition, we contribute various optimizations to speed up the performance. For instance, we implement the most frequently executed states of the TDFA as fast *executable* code on the IXP cores. As (T)DFAs may well be too large to fit in the instruction buffer of a core, we place the remaining states in slower off-chip memory from where they are accessed by a (T)DFA *interpreter*. Such a hybrid approach yields very good performance for the most common patterns and great flexibility for less common ones. Finally, use of the language is extremely friendly. The device is incorporated as a normal network card in the Linux system and executing new programs on the card is as simple as compiling and executing a high-level language on a normal CPU.

Although this paper mainly discusses the IXP2xxx implementation, to demonstrate its portability we also implemented Ruler back-ends to generate Verilog for FPGAs, and ANSI C code for general-purpose processors[1]. We will show that even older and fairly low-end IXPs can achieve performance that makes them applicable to gigabit networks.

This paper is organized as follows. Section 2 presents the Ruler language for packet matching and rewriting. We present the implementation in Section 3 and the details on the IXP2xxx in Section 4. Performance is evaluated in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2. THE RULER LANGUAGE

Ruler is a language to match and rewrite network packets. It was originally developed for packet anonymization, but because of its flexibility it can be used in a wide range of other applications, including data collection and reduction, but also traffic filtering and classification for intrusion prevention systems like [8].

A Ruler program is called a *filter*. A filter specifies rewrite rules that contain *patterns* that are matched against sequences of bytes. If a sequence matches a pattern, the filter applies an operation on the sequence. The operation may be to accept or reject the sequence, but rewriting to a new sequence is also supported. The combination of a match pattern and an operation is called a *rule*. In principle an input sequence can come from any source, but in the context of this paper it is a network packet. For example, the following filter:

[1]A report about Ruler on x86 processors, as well as its implementation and all the backends are available from
`http://projects.gforge.cs.vu.nl/ruler/`

```
filter udp
  h:(byte#12 0x800~2 byte#9 17 byte#2)
  a:(192 168 1 byte)
  t:*
    =>
  h 0#4 t;
```

matches 12 bytes with arbitrary values, the constant `0x800` as the next two bytes (the length is indicated by `~2`), another nine arbitrary bytes, the constant byte 17, another two arbitrary bytes, a sequence of three bytes with specific values, a single arbitrary byte, and a sequence of zero or more arbitrary bytes after that. Assuming we are matching Ethernet packets, the bytes up to and including `0x800` match an Ethernet header with the protocol number for the IP protocol, while the remaining patterns match a UDP packet with the sender IP address in the range 192.168.1.xx and arbitrary values for the other bytes. The bytes of the IP address are given the label `a`, the bytes before `a` are labeled `h`, and the bytes after `a` are labeled `t`. If a packet matches this pattern, it is rewritten to a new packet with the same contents for the areas covered by `h` and `t`, but four zeroes at the place of `a`. Thus, this rule implements a simple anonymization on UDP packets by zeroing the sender IP address.

A filter can contain multiple rules. The order of the rules is important: the first rule that matches applies. If no rule matches, Ruler rejects the packet by default, but the user can easily change this behavior by adding a rule that accepts any packet.

2.1 Predefined patterns

It is error-prone and tedious to explicitly specify the layout of all headers in all filters, so Ruler supports reusable pattern definitions. Also, individual fields of the pattern can be substituted to specialize a pattern. For example, the following filter first defines a pattern for the Ethernet protocol header, and then defines a specialized version that only matches Ethernet headers with the IPv4 protocol number. Notice the use of the `with` construct to specialize the pattern, and to replace the source address in the result pattern:

```
pattern Ethernet:
    (dst:byte#6 src:byte#6 proto:byte#2)

pattern Ethernet_IPv4:
    Ethernet with [proto = 0x0800~2]

filter ether
  e:Ethernet_IPv4 t:*
    =>
  e with [src=0#6] t;
```

As a larger example, the following is the layout definition of the header of the IP version 4 protocol:

```
pattern IPv4: (
    { version:4~4 ihl:bit#4 }
    tos: { precedence:bit#3
        D:bit T:bit R:bit bit bit }
    iplength: byte#2
    id: byte#2
    { fragment_flags:{ R:bit DF:bit MF:bit }
      fragment_offset:bit#13 }
    ttl: byte
    protocol: byte
    check: byte#2
    ipsrc: byte#4
    ipdest: byte#4
    options: byte#(4*ihl-20)
)
```

The byte and bit order of the Ruler pattern specifications were chosen to correspond with those of the RFC specifications of the TCP/IP protocol family. Thus, our IPv4 layout closely resembles the IP header specification of RFC 791.

This example also shows another feature of Ruler patterns: *bit patterns*. Patterns within curly brackets ('{}') define a pattern at bit level instead of byte level. Thus, `fragment_flags` refers to three bits in the IP header, while `fragment_flags.MF` selects a single bit from this group. The details of bit patterns are beyond the scope of this paper. Since Ruler requires that bit patterns surrounded by byte patterns match a known multiple of 8 bits, we can treat them here as 'syntactic sugar' for specifying byte patterns.

Another issue is that the size of an IP protocol header is variable. Specifically, the `ihl` field in that header specifies how many 4-byte words the header contains. To support this construct, Ruler state machines use a special *memory-inspection* state that transitions to a next state depending on the value of a specific range of bits in memory. For example, the `options` field in the pattern above has a length that is dependent on the value of the `ihl` field.

2.2 File inclusion

Ruler also supports file inclusion. Together with the predefined patterns this allows common layouts to be defined once and used in many filters. The following filter uses two layouts from the standard Ruler layouts definition file `layouts.rli`. The shown filter only accepts IPv4 packets that have the IPv4 flag `MF` set.

```
include "layouts.rli"

filter fragment
  Ethernet_IPv4 IPv4 with [fragment_flags.MF=1] *
    =>
  accept;
```

Notice that `accept` is used as result pattern. This indicates that the result packet is the same as the input packet. There is also a `reject` result pattern that indicates the packet should be rejected.

2.3 Classification codes

Finally, a rule can give a classification code to a matched packet. For example, in the filter:

```
pattern Ether: (dst:byte#6 src:byte#6 proto:byte#2)

filter ether
  Ether with [proto=0x0800~2] * => accept class 1;
  Ether with [proto=0x0806~2] * => accept class 2;
```

the first rule matches packets with the IPv4 protocol number, and assigns them class 1, and the second rule matches packets with the ARP protocol number, and assigns them class 2. In the IXP implementation of Ruler the class numbers determine the disposition of the packets: we send them back to the network, to the host system, or both. However, the class number could carry more information. For instance, on devices with multiple ports, Ruler could use class numbers to route the packets.

3. RULER IMPLEMENTATION

Since Ruler is intended for processing high-speed network links, we must pay close attention to the performance of the implementation. We use the traditional approach for high-speed pattern matching by constructing a Deterministic Finite Automaton (DFA), but Ruler poses a significant complication because it not only matches patterns, but also uses fragments of the matched pattern to construct the result pattern. Traditional matching algorithms do not record these positions, but we will describe a method to do this.

3.1 Parallel pattern matching

For efficiency reasons it is essential to examine each byte of the input sequence only once, and never backtrack, even if the input matches multiple patterns or matches a pattern in multiple ways. This is a well-known problem, and we use the traditional solution by constructing first a Nondeterministic Finite Automaton (NFA) and from that a Deterministic Finite Automaton (DFA). In this section we briefly describe the construction process. In the next section we show an example that also covers our extensions to this construction process.

The NFA is a straightforward translation of the match patterns: each state in the machine indicates that a particular fragment of the input pattern has been matched, and the transitions between the states are labeled with the values that must be matched to reach the next state. Thus, a string matches the entire pattern if there is a matching sequence of transitions through the NFA that ends in an accepting state at the end of the input. The matching process is described by a **non**-deterministic automaton because the match pattern may contain choices: alternative patterns or variable-length patterns.

Since matching against an NFA on standard processors is not efficient, we again follow the conventional implementation process for matcher state machines, and construct a Deterministic Finite Automaton from the NFA through the so-called *subset* algorithm. We represent each distinct set of reachable NFA states as a DFA state. The transitions from a DFA state are computed by enumerating all possible transition values, and taking the union of the transitions of the NFA states of that particular DFA state for a transition value. The resulting union of NFA states represents another DFA state, or, for an empty set, a non-existent transition. If one of the NFA states is an accept state, the DFA is also an accept state. Since we know the start state of the DFA (the set containing only the NFA start state), we can compute the entire DFA by systematically following all possible transitions from all possible states. In the worst case, an NFA with n states can result in a DFA with 2^n states, but in practice the number of states is much lower.

To match a string against the DFA, the bytes in the string are examined one by one, and the transition that matches the value is taken. If there is no such transition, the string does not match. At the end of the input the DFA should be in an accepting state. In some cases it is clear that the input will match before the final character is examined, typically because the DFA has entered an accepting state that only transitions to itself. Stopping the matching process early in such a state is an obvious optimization.

3.2 Matching with tags

The traditional translation scheme of the previous section leaves a significant problem unsolved: Ruler uses parts of the matched input sequence in the output sequence. Therefore, the matching process must also record the start and end position of these parts. For example, for the filter

```
filter byte42
  * 42 b:(byte 42) * => b;
```

the start and end position of pattern `b` must be determined, since `b` is used as result pattern. The traditional match process only determines whether a string matches the entire pattern or not, it does not yield intermediate positions. Fortunately, Laurikari [19] describes a generalization of DFAs that handles this. As far as we know, Ruler is the first program to apply this theory to network processing. It allows us to not only do high-speed packet matching, but also high-speed packet rewriting.

After Laurikari, we introduce a new type of state, a *tag* state. Fig. 2a shows an NFA to match the input pattern of filter `byte42`. State 1, represented as a square, records the current position in the input in a memory location called a *tag*; in this case in tag 0. As usual, an NFA matches an input if a sequence of transitions through the NFA can be found that match the input. The sequence should start in the start state, here state 0, and end in the accepting state,

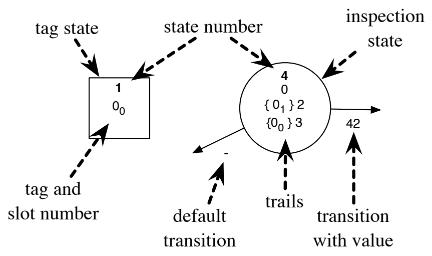

Figure 1: The meaning of the various elements in the state diagrams of Fig. 2.

here state 4. The sequence of transitions will pass through state 1, which records the start of pattern `b`. The end of pattern `b` is not explicitly recorded, but is 2 bytes further[2].

As before, matching a string requires finding a sequence of NFA transitions that matches the input characters and that ends in an accepting state of the NFA. However, since we are also interested in the positions of the tags, we must also keep track of the possible values for the tags for possible matches. For example, for the NFA of Fig. 2a and the byte sequence `42 42 42` the sequence is

$$0 \xrightarrow{42} 1 \rightarrow 2 \xrightarrow{?} 3 \xrightarrow{42} 4$$

We pass tag state 1 after one character has been matched, so the tag is put at position 1. For the string `42 42 0 42` the sequence is

$$0 \xrightarrow{?} 0 \xrightarrow{42} 1 \rightarrow 2 \xrightarrow{?} 3 \xrightarrow{42} 4$$

and the tag is put at position 2. As these examples indicate, different transition sequences can pass a tag state at different positions in the match string.

The DFA that we generate must of course yield the correct value for every possible input string. This means that during the matching process it may have to keep track of multiple *potential* values for a tag position. For example, for the NFA of Fig. 2a and the partial input `42 42`, there are two potential tag positions, depending on the subsequent input. Only when the match is completed can we decide which underlying NFA state determines the match, and consequently which set of potential tag positions applies. Therefore, it is no longer sufficient to represent a DFA state as a set of NFA state numbers; we must also remember the potential tag positions that were recorded before an NFA state was reached. To do this, we maintain for each tag a set of storage locations, called *slots*, that contain earlier stored tag locations.

Despite the complications, we can still apply the subset algorithm. However, a DFA state now does not consist of just a set of NFA states, but for each NFA state we must keep track of the tag slots where its positions are recorded. We call such a combination of NFA state and tag slots a *trail*, and each DFA state consists of a *set* of trails.

When we compute the destination state of an outgoing transition, we may encounter a tag state in the NFA. If we do, we follow the outgoing transition of the tag state, and add a new tag slot to that particular trail. We also insert a tag state in the DFA to fill the tag slot. Since a DFA state can contain multiple trails, and since every outgoing transition of an NFA state may be to a tag state, the computation for the destination state may encounter multiple tags. All these tags can be collected in one tag state in the DFA.

[2]For didactical reasons this example uses a tag state to record the *start* of the fixed-length pattern. In reality we tag the *end* of such a pattern, since that results in smaller machines.

The slot to fill for each tag is chosen by examining the slots of the other trails in a DFA state: the lowest-numbered one that is not used is assigned to the tag position. A newly constructed DFA state is equal to an earlier one if it has the same set of trails. A newly constructed tag state in the DFA is equal to a previous one if it contains the same tag slots, and transitions to the same state.

For example, Fig. 2b shows the result of applying the subset algorithm to the NFA of Fig. 2a. Figure 1 shows the meaning of the various components in the DFA.

3.3 Rewrite patterns

Once a pattern is matched, an output byte sequence must be constructed. During the match process various potential values for the match locations were recorded, but in some cases the real match location values to use are only known once we have arrived at an accepting state.

An output pattern is constructed by copying fragments from the input pattern and/or introducing new byte sequences. This operation is expensive, because all the bytes of the result have to be moved into place. Ruler avoids this operation if the filter only contains `accept` and `reject` rules. Ruler can also largely avoid this operation if the output pattern only truncates and/or modifies a few bytes of the input.

3.4 State types

All in all, Ruler DFAs may contain the following types of states:

- **Inspection states** examine a byte in the input text, and make a transition to a next state depending on the value of the byte.
- **Jump states** skip a given number of bytes in the input text without inspecting them, and make a transition to a fixed next state. They are useful for jumping directly to interesting data (skipping unrelated headers or payload prefixes).
- **Memory-inspection states** inspect a value earlier in the input, and make a transition to a next state depending on the value. They are essential for handling offsets that depend on fields in the data such as the length of IP or TCP headers).
- **Tag states** record the current position in the input into one or more tag slots, and make a transition to a fixed next state.
- **Accepting states** terminate the matching process, and fill the final tag values from one set of tag slots.

4. RULER ON NETWORK PROCESSORS

In this section, we explain in detail the implementation of Ruler on the Intel IXP2xxx series of NPUs. To familiarize the reader with the relevant architectural features, we first briefly introduce the IXP architecture (Section 4.1) and then describe how we used the hardware to achieve an efficient implementation of our language (Section 4.2).

4.1 The Intel IXP2xxx series

Intel IXP2xxx network processors are heterogeneous multi-cores with various hardware assists (e.g, hashing or crypto units) and different levels of memory. The cores support multiple hardware threads with zero-cycle context switching to allow for memory latency hiding. While we explicitly target IXPs, multi-core architectures are quite common in this field and processors like the IBM PowerNP, Agere Payload Plus, Xelerator X11, and Cell share many of the characteristics of the IXP.

The IXP2xxx [13] is a family of NPUs with an Intel *XScale* [14] core (ARMv5 compliant) and a number of programmable execution

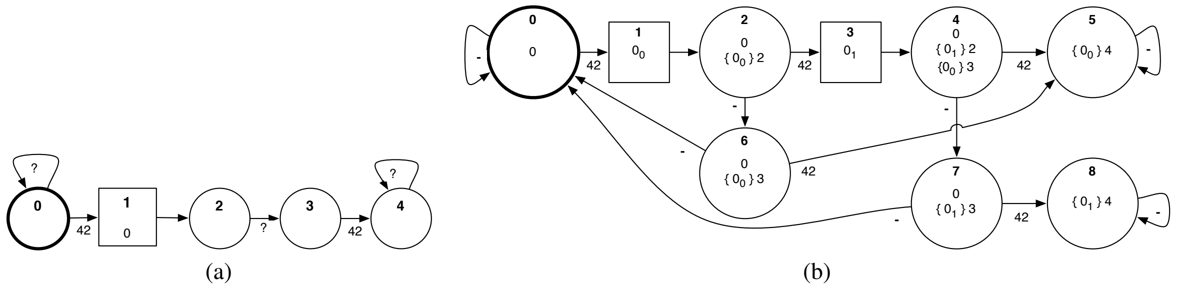

Figure 2: Tagged NFA (a) and DFA (b) for pattern `* 42 b:(byte 42) *`. See Fig. 1 for the meaning of the various components.

units known as microengines (MEs) tailored for packet processing. Depending on the model, it may have 8 (IXP2400, 600MHz), 16 (IXP28xx, 1.4GHz) or 4 (IXP2350, 900MHz) of such MEs. Every ME runs an independent program from its own instruction store. Depending on the IXP model, the instruction store has 4K (IXP2400) or 8K (IXP2800) slots for instructions. In addition to such core-level parallelism, each ME offers 8 hardware contexts (threads) with zero-cycle context switching and asynchronous memory access. Each context has its own set of registers and a program counter, but they share the instruction store. IXPs offer various kinds of memory ranging from fast but small memory which is local for each ME, to larger and slower ones. Typically, the ME's local memory has 640 4B words, while shared on-chip Scratch space is 16KB. Off-chip SRAM is 8-16MB, and off-chip DRAM is 256MB-1GB. MEs are connected into a chain and neighboring engines can share registers. This enables them to pass data without going off-chip to memory. Finally, the IXPs offer hardware support for expensive computations like hashing, CRC computation, encryption, and decryption.

Hiding memory latency. A key difference between MEs and ordinary CPUs is asynchronous memory access. Memory latency is a well-known issue and caching is commonly used to avoid performance loss due to lengthy transfers from main memory. In network processing, data is often touched only once, so caching has limited effect. On the other hand, access patterns are often predictable and prefetching helps. IXPs offer asynchronous memory access, which allows cores to continue executing instructions (either in the same thread or a different one), while waiting for a *set* of transfers to complete.

While powerful, asynchronous memory access with a hierarchy of memories of different speeds and sizes exposes all *"caching"* complexity to the programmers. They are responsible for decisions such as when to place what data structures in which memory.

Non-reordering locks. In packet processing, it is important not to reorder traffic. Some protocols like TCP can handle out of order arrival of packets. However, it reduces the speed of a connection. For other protocols, like real-time audio/video, keeping the packets in order is critical, since the end point can rarely wait until the out-of-order packets arrive. Consequently, we need to avoid reordering threads that handle packets, which rules out any waiting queues (linked lists) were we can put threads to wait for an event to happen. For this purpose, we use two atomic counters to implement mutual exclusion similar to the Lamport's bakery algorithm [18]. Both counters are initiated with the same value, one hands out tickets to threads, the other says which tread may enter the critical section.

Rings. A final hardware feature that is key for performance is the IXP's support for rings. Rings can be used to store and retrieve elements without need of locking. They are supported by both Scratch and SRAM. In most practical implementations involving IXPs, including ours, one ME is dedicated to receiving packets (RX), one is reserved for transmitting packets (TX), while the other MEs do the actual packet processing. In other words at least 3 MEs touch each packet. The hardware rings serve the inter-thread communication. A test-and-increment operation allows us to acquire a slot in a ring buffer atomically. When a thread wants to get a slot, it reads the actual write pointer and increments the value for another thread. Atomicity of the *put* and *get* operations immediately synchronizes all accessing threads and so it keeps data queues consistent.

4.2 Ruler on the IXP2xxx

The Ruler implementation on IXP consists of several components. As Fig. 3 shows, each packet enters the system via a dedicated RX engine that receives packets from the network and stores them to DRAM. The core of the system are the Ruler engines that run the pattern matching and packet rewriting rules. If a packet is not dropped, the TX engine transmits it to the network and/or sends it to the host system. It is the single point of routing to multiple ports and PCI. Having separate RX and TX engines avoids complex synchronization issues.

Each Ruler engine runs one or more threads that execute the code of the DFA matcher. The DFA states are encoded in instruction store or optionally interpreted from memory. There is no cooperation between the threads; each of them works on a different packet. When a thread on a Ruler engine retrieves a packet enqueued by the RX, it starts feeding bytes to the matcher until a match is found or until it consumes the entire packet. In the former case, it performs the corresponding rewrite action of the selected rule; in the latter, a default end-of-packet action.

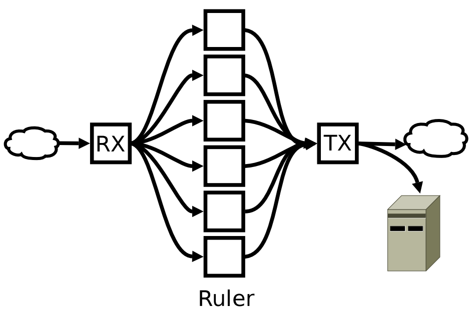

Figure 3: The main Ruler components on the IXP. Each square stands for an ME and arrows show the path a packet takes.

4.3 Reading data

The Ruler DFA matcher always works on single bytes. Therefore, the DFA engine always receives the packet one byte at a time. However, the IXP memory controller is tuned for large, aligned transfers rather than for reading/writing single bytes. For instance, it is only possible to transfer 8B chunks aligned to 8B boundaries between IXP registers and DRAM.

Our solution is to prefetch asynchronously 64 bytes to a buffer in the ME's local memory and satisfy requests for each new byte from this buffer while a new request for a read from DRAM to the transfer registers is pending. Reading the next 64 bytes from DRAM takes less time than inspecting the whole current buffer, so we completely hide the DRAM latency, and a single thread is sufficient to supply the data for the entire ME.

Every call to the routine that returns the next byte takes 15 cycles. Since the fastest inspection operation takes 3 cycles, the DFA may spend 83% of its time in this routine. Inlining the routine saves 4 cycles, and boosts the speed of the fastest inspection state by 22.2%. However, inlining requires more instructions and thus fewer states can fit in the instruction store, so we made this a compile-time option.

For packet headers we employ another optimization. As long as we are examining bytes with a fixed offset from the start of the packet, we can access them in only two clock cycles as follows: we prepare a mask to read one byte from a four byte memory cell, and then we read the byte. This reduces the original overhead of 15 cycles to only 2 and also significantly reduces code size. Moreover, since the header bytes are the first to be inspected in every packet, and since they typically are inspected to select a particular class of packets, all packets outside that class can be rejected without ever executing more expensive states.

4.4 Inspection states

Unlike getting a new byte, which is a constant time operation, the number of cycles required for inspecting that byte varies very much. Basically, determining what action to take based on the input byte is a multi-way branch much like the `switch` statement. A naive implementation that compares against each possible value one by one is expensive in both code size and execution time. Therefore, modern compilers try to generate more efficient code. Examples include implementations based on hash tables [11] or trees [10]. Our IXP code generator implements several algorithms and has infrastructure to add more. Since we do not have enough fast memory to store a hash table for each state, we decided to implement a binary tree in the instruction store with several balancing policies in addition to the naive implementation. We will now discuss the main method we used to optimize the switch statements.

The IXP instruction set provides `br=byte` and `br!=byte` instructions that compare a selected byte out of 4 in a 32-bit register to an immediate value in a single cycle and jumps to a label if the byte *is* or *is not* equal the value. We know how many cycles an instruction costs, therefore we can compute how many cycles are required for each value of the input byte for each *switch* implementation. We choose the one that needs the smallest weighted mean of cycles to decide the jump to the next state. In our experience, the default branch of a switch statement is the one taken most frequently. This makes intuitive sense also: a DFA usually spends most of its time matching the packet payload against a number of specific patterns. In most cases the current byte is not the expected one and the input does not match any filter pattern. Unfortunately, the default is also the most expensive branch since it is taken only if all other tested options were rejected. The best balancing policy for binary trees that we use is to optimize the weighted mean of cycles that an ME spends in the state when the default branch is taken.

In each node one bit of the input byte is checked to decide which of the two subtrees to search. The IXP has instructions `br_bset` and `br_bclr` to test the value of a single bit in a 32-bit register in a single cycle and jump if the condition holds. We select the bit to test depending on its ability to separate the input values that result to the default jump in one branch and the other values to the other branch. However, it is not always possible to pick such a bit, therefore the bit with the highest difference between the number of defaults in both branches is chosen. The branch with the highest probability that leads to the default jump is implemented as the cheaper one, i.e., the execution does not jump to this branch, but falls through the test-bit-jump instruction. Following this branch of the tree takes 1 cycle; taking the other branch takes 4 cycles. For instance, if the default branch applies to all values greater than 127 and the most significant bit of the byte is set, we don't need any more checks and the tree handles 128 values in a single cycle. This is quite a common case.

Since the input value is a single byte, since we always check a single bit, and since we never check a bit more than once, the depth of the tree is at most 8 levels. However, a complete binary tree with 8 levels would be unreasonably large and slow. For this reason, once we have a branch with no jumps to a default target, we fall back to the naive switch for this subtree.

Note that in general the binary trees require more instructions then the naive implementation and therefore fewer states will fit in the instruction store. Still, taking everything into consideration, we measured up to 10% speedup in real traffic. The trade-off between speedup and the number of instructions is discussed in Section 5.2.

4.5 Tag states

As explained in Sec. 3.2, during the match process we may have to store a number of potential tag positions. As with all other states in the state machine, we must be careful to provide an efficient implementation of these tag states. We store a table in the *local memory*. Local memory is tiny, but fast for contiguous reads and writes. A tag state may write the same value to more than one tag (but to only one slot for each tag) simultaneously. Therefore we organize the slots in the table in such a way that the state updates neighboring locations without losing cycles on resetting the index registers. Since 16 bits is enough to store a tag, we store two tags in a single memory cell. Therefore two tags may be written at once and the neighboring one immediately in the next cycle.

4.6 Executed vs. Interpreted states

The number of DFA states can easily be larger than can fit in the small instruction store of an IXP's ME. Code cannot be executed from any other memory. Swapping code between instruction store and memory is possible, but too slow for the data-path. Therefore we have opted for a hybrid model where we implement the frequently executed (hot) states in IXP instructions and place the remaining states as a table into SRAM or Scratch memory. These states are executed by a small interpreter. We present our measurements on the hot states in Section 5.1.

When we generate the IXP code, we first generate code for all states of the DFA, and compute how many instructions are consumed by the entire DFA together with support routines. If necessary, we move states of the DFA to memory, taking into account that if we do so, an interpreter must be also included in the instruction store. For every transition from an *executed* state to an *interpreted* state we need a *stub* state that starts the interpreter in the correct state. Similarly, the interpreter must be able to return execution to a state in the instruction store if necessary.

To decide which memory states must be moved to memory, we assume that the most frequently executed (the *hottest*) states are close to the start state (see Section 5.1). Therefore we first try to move the most distant states to SRAM to keep the closest states in instruction store. To implement this, we assign a distance to each state of the DFA which represents the number of hops on the shortest path from the start state. The algorithm iterates over states, working from the most distant ones back to the start state. It moves states to memory until the number of instructions is reduced enough to fit in the instruction store, adding or deleting stub states as necessary.

At first glance, it might look like the *interpreted* states must be much slower. However, the effect is reduced if the memory latency is hidden by executing other code. When multiple threads are running, execution of an *interpreted* state takes at most **35 cycles** including getting the next byte from the packet. The interpreter reads the state type, the next transition and unpacks the address of the next state and the flag that signals whether it is *interpreted* or *executed*. It needs at most 2 reads from memory. Since the latency is hidden by execution of other threads, we account each read as a single-cycle operation. 35 cycles is about **28%** more than consumed by the average states that we observed in practice that were implemented by instructions. Moreover, in an *interpreted* state, the number of cycles does not depend on the complexity of the state, i.e. the number of outgoing edges. Therefore, if the estimated number of cycles of a state in the instruction store exceeds the threshold of 35 cycles, we make it an *interpreted* state too. The estimation is done for each inspection state when deciding about the implementation of the switch statement (Section 4.4).

4.7 Packet rewriting

To support rewrite rules, we implemented an interpreter of the rewrite patterns. We represent a rewrite pattern as a list of descriptors, each of which either copies a substring from the original packet, or inserts a new string. Descriptors are stored in SRAM.

In many cases in-place rewriting is complicated and slow. Note that when we copy a byte from the original packet to a new unaligned location within the packet and insert another new byte next to it, we need to transfer the same 8 bytes at least twice from and back to memory in the read-modify-update way. Some actions can produce even more complicated memory usage. Therefore we opted for assembling the *new* packet in local memory buffer. Above all, we benefit from the local memory's access time which is almost as good as that of registers and it is possible to use it with `byte_align_be` and `byte_align_le` instructions for concatenation of unaligned data in two 4 byte registers. First, we read a few bytes from DRAM (copy) or SRAM (insert) and we use the align-instructions to shift the possibly unaligned data and save them in a small temporary buffer. In the second step, we append the temporary buffer to the new packet in local memory. We do not support modification on the bit level. After finishing the new packet, we write it back to its original location in DRAM in 32-byte asynchronous bursts. While 32 bytes are in transfer, we prepare the next 32 bytes. Unfortunately, the local memory has room for only a single buffer for constructing new packets, so for rewrite rules we must run Ruler in single-threaded mode. As long as the matcher fits into the instruction store, the matching performance is not affected. This is generally the case for packet header anonymization.

4.8 Host communication

In our experiments we use the ENP2611 board, a PCI card with an IXP2400. To make Ruler on the IXP available to any packet processing framework on Linux, we developed an Ethernet device driver for Linux 2.6 kernels. The driver takes full advantage of DMA by the TX micro engine and the NAPI [15] design for interrupt handling. To make the programming of the device as simple as possible, the driver supports the standard Linux mechanism to upload firmware: it exports entries in the special directory `/sys/firmware`. The user writes the compiled Ruler filter to a file in this directory to reprogram the board. When the board is then reset (`ifconfig` *dev* `up`), the new firmware is loaded. The user can also read statistics from `/sys/firmware`, such as the number of received, dropped and processed packets.

5. PERFORMANCE EVALUATION

In this section, we first evaluate the locality in DFAs that motivates the hybrid *executed-interpreted* approach. Next, we evaluate our switch implementation. Finally, we perform throughput measurements on a variety of filters.

5.1 Locality in DFAs and Interpreted states

An advantage of the *interpreted* states is that the execution time per byte is constant. However, for all but the most complex states, executing the state is faster. Therefore we try to keep the hot states in the instruction store and we try to place only those states in SRAM that do not fit in instruction store and are rarely executed.

However, doing so only makes sense if there is a set of hot states in the DFA. To support the heuristics that we use, we therefore conducted a series of experiments. We converted several sets of Snort [21] rules (e.g., web, pop3) to Ruler filters and collected statistics about the number of times states were visited on a workstation. The Snort rules we used first inspect the packet headers by looking at some of the fields and skipping the others. Because we know the positions at compile time, we can optimize this part of the code more than the payload scanning part. The payload is then typically inspected by a pattern that begins with a star-loop, so that we may find the match at any position.

Number of rules	46
Number of states	**227**
Number of executed states	**46**
Weighted mean size of a state†	14.62
Weighted mean branches inspected	9.77
Maximal executed level	13
Executed states (in instruction store)	173
Interpreted states (in memory)	54
Memory entries	12
Instructions saved by *interpreted* states	1268
Used instructions	4082
Unused instruction store slots	6

†Weighted mean size: *weighted* means how many times a states was executed, while *size* refers to the number of switch statement branches

Table 1: Statistics collected for the Snort *web-attacks* set of rules on an office workstation.

Table 1 presents measurements for the *web-attacks* set of rules. The results for other sets are similar. The first 10 states inspect the packet headers (ETH, IP, TCP), each of them is executed only once per packet, in total they consume **6%** of the execution time, the payload inspection spends **69%** of the execution time in the star-loop head state, clearly because there is normally no interesting pattern in the payload. The remaining **25%** of time is consumed by states at levels 11, 12 and 13. Execution reaches states on deeper levels only in the rare case that a match is found. We conclude that accesses to our DFAs exhibit strong locality and storing the hot first few levels helps.

5.2 Switch statements

As stated before, we have several strategies for implementing switch statements and the binary trees speed up processing by up to 10% compared to the trivial implementation. For these results, we do not make any assumption about the traffic distribution, we only promote the default branch. Some states show a slowdown because they are not executed so frequently. Fortunately, the time the DFA spends in such states is negligible compared to the hot ones and their speedup outweighs the slowdown. We observed that the states that have a major impact on the performance are the heads of the star-loops. Indeed, their speedup is almost 10% of the overall performance. The binary trees require more instruction store, therefore we must implement more states as *interpreted*. In practice, however, no *hot* states were ever pushed to memory, so the effect was quite limited.

5.3 Benchmark runs

We carried out several performance tests to determine the maximum achievable throughput of our Ruler implementation. We deliberately evaluated Ruler performance on an NPU that is not top of the line (an older IXP2400 geared for gigabit performance) for two reasons. First, we aim to show that Ruler is suitable for low-cost network devices[3]. Second, by opting for the low-end IXP2400 instead of, say, the IXDP2850, we were better able to assess the bottlenecks and limitations of our language and implementation. In contrast, for our IXPDP2850 we were not even able to generate sufficient traffic to saturate the processor's capacity. In all benchmarks we use the inlined version of the byte fetching routines (Sec. 4.3) and the optimized switch implementation (Sec. 4.4).

Ideally, a filter should be able to handle any volume and mixture of traffic on the Gigabit network link of the board, but in practice this is not possible for all Ruler filters. To explore the performance properties of the IXP Ruler filters, we have evaluated a number of different filters that highlight different aspects of the IXP Ruler implementation. The simplest one is the null **null** filter, which accepts all packets and serves as a 'baseline' filter. We also benchmark two variants of a typical anonymization filter that matches all IP packets and zeroes the source and destination address; **anonym** retains the payload (and therefore only modifies eight bytes in the header), and **anonymhdr** drops all bytes beyond the IP header. The filter **matchpayload** rejects all IP packets that contain the string `REJECT` anywhere in the payload, and accepts all other packets. This filter has to inspect all bytes of a packet that doesn't contain this string. Finally, we include two large filters in the benchmark set: filter **backdoor** matches against 90 rules in the 'backdoor' set of the Snort patterns, and filter the **large** which is defined as follows:

```
include "layouts.rli"
filter large
  eh:Ethernet_IPv4 ih:IPv4 uh:UDP
  * (0x00 byte#6 "z") *
    => reject;
  eh:Ethernet_IPv4 ih:IPv4 uh:UDP
  * (0xff byte#6 0x11) *
    => reject;
  eh:Ethernet_IPv4 ih:IPv4 uh:UDP
  * ("R" byte#4 "T") *
    => reject;
  * => accept;
```

This filter is rather artificial, but requires a large number of states because when the start of a pattern is encountered, here `0x00`, `0xff` or `"R"`, the DFA has to ensure the correct end byte, here `"z"`, `0x11`, resp. `"T"`, occurs at the correct place. Meanwhile, the DFA has to keep track of all other potential starts of these patterns, which requires a large number of states.

[3]Cheap in manufacturing cost, not necessarily in retail prices.

filter	states	instructions	insns/state	interpreted states
anonym	19	641	30.05	
anonymhdr	19	641	30.05	
backdoor	2441	46041	18.83	2147
large	2327	19216	8.23	2141
matchpayload	24	400	13.75	
null	1	145	6.00	

Table 2: The sizes of the benchmark filters. The second column shows the size of the entire IXP program, the next column tells how many instructions we use to encode a state on average, and the final column indicates how many states (if any) are represented as table rows.

We expect the **backdoor** filter to adhere to the observations of Sec. 5.1, and have a few very 'hot' states that indicate no suspect patterns have been found yet, and a large number of very 'cold' states that are only entered when a large part of a suspect pattern has been matched. In contrast, in the **large** filter the distinction between hot and cold states is not so clear, so our heuristics are not as effective in this case. Table 2 lists the number of states and the number of IXP instructions needed to implement all of the filters. Both **backdoor** and **large** do not fit in the instruction store and need interpreted states, see Sec. 4.6.

We subjected all these filters to two classes of network traffic: synthetic 'worst case' traffic, and replayed 'normal' network traffic. In both cases we saturate the 1Gbit/s network link as far as technically possible. Note that the traffic generator in our experiment is not sending packets at a constant rate, but rather in bursts, so it is not safe to extrapolate results below 1Gbit/s to full 1Gbit/s.

For the synthetic traffic we repeatedly transmit the same UDP packet at such a fast rate that we (as far as possible) saturate a 1Gbit/s Ethernet link. To show the load the various filters put on the processors, we run the filter on different numbers of MEs, and show in each case the percentage of packets that cannot be processed by the MEs, and have to be dropped. Table 3 shows the results for these experiments.

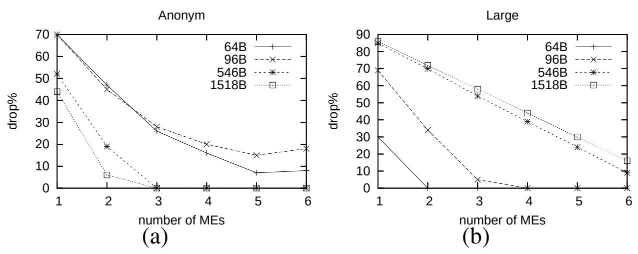

Figure 4: The effect of adding MEs to the `anonym` and `large` filters.

For two of the filters the processor is not always able to process all traffic, even with all MEs enabled. We show the results of these two cases graphically in Fig. 4. In a sense, `anonym` and `large` exhibit opposite behavior. The **anonym** filter (Fig. 4(a)) is too slow for small packets since there is a noticeable overhead for each received packet. At a certain point, adding more MEs does not help, indicating that the bottleneck is presumably in the access to SRAM or DRAM. Note that the **anonymhdr** filter, which is similar to **anonym** but does not output the payload of the packet, can easily keep up with the traffic rate. Again this indicates that the construction of the rewritten packet in single-thread mode causes the

	packet size 64						packet size 96						packet size 546						packet size 1518					
	531.9 Mbit/s						751.2 Mbit/s						962.1 Mbit/s						990.8 Mbit/s					
	1	**2**	**3**	**4**	**5**	**6**	**1**	**2**	**3**	**4**	**5**	**6**	**1**	**2**	**3**	**4**	**5**	**6**	**1**	**2**	**3**	**4**	**5**	**6**
anonym	70	47	26	16	7	8	70	45	28	20	15	18	52	19	0	0	0	0	44	6	0	0	0	0
anonymhdr	55	18	3	0	0	0	52	17	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
backdoor	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
large	30	0	0	0	0	0	69	34	5	0	0	0	85	70	54	39	24	9	86	72	58	44	30	16
matchpayload	3	0	0	0	0	0	50	0	0	0	0	0	71	43	14	0	0	0	73	46	20	0	0	0
null	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 3: Percentages of dropped packets for various input packet sizes, numbers of active MEs, and for different filters. For example, the anonym filter running with 3 MEs drops 28% of 96-byte packets. Note that our traffic generators are not always able to generate a full 1Gbit/s. The actual rate is shown in the table.

slowdown. If the filter only rewrites a small fraction of the packets, threads can run in multi-threaded mode, reuse the assembly buffer, and serialize in case of conflict.

The **large** filter (Fig. 4(b)) is too slow for some inputs because its DFA has to be implemented partially in interpreted states, and since it has no clear 'hot' states, it spends a relatively large part of its time in these interpreted states. The problem mainly shows for large packet sizes because there a larger fraction of the received bytes is part of the payload, and is not processed by the faster states that process the header.

	829.0 Mbit/s					
	av. pkt size 305.0					
	1	**2**	**3**	**4**	**5**	**6**
anonym	78	37	2	0	0	0
anonymhdr	3	0	0	0	0	0
backdoor	0	0	0	0	0	0
large	82	59	31	10	3	1
matchpayload	79	48	13	0	0	0

Table 4: Percentages of dropped packets for network traces, for different numbers of active MEs, and for different filters.

To test our system on more realistic traffic we ran the same filters on a pre-recorded data stream from the 1998 DARPA Intrusion Detection Evaluation Data Set [6, 7]. In particular, we used the trace file for Monday of the first week of training data, a 266 Mbyte trace file. We ignored the timing information in the traces, and replayed the traffic as fast as possible. Again we ran the experiment for different numbers of MEs. The results are shown in Table 4. All filters could handle the traffic with 6 MEs without significant packet loss.

6. RELATED WORK

The need for deep packet inspection in monitoring and classification of live traffic is increasing (e.g., for security, QoS, HTTP load balancing, traffic engineering, anonymization, NAT, network debugging, filtering and other applications). Many systems base the classification on matching sets of rules. In embedded devices, matches are mostly based on simple byte patterns often using (modified versions of) well-known solutions such as the Aho-Corasick algorithm [1] (examples include [2,3]), or content-addressable memory [24]. Some more advanced systems also offer regular expressions (e.g., [4] and this paper). However, to our knowledge, we are the first to offer rewriting based on regular expressions.

Using DFAs for efficient pattern matching is quite common. However, with increasing number of rules the size of the corresponding DFAs grows quickly, so that more slow off-chip memory is required. Efficient implementation of DFAs on both standard and network processors is still an active research area. For example, in [23] it is shown that common patterns for packet inspection can be transformed in such a way that the DFA does not grow exponentially. Kumar et al [16] introduce a technique that essentially allows multiple states to share part of their next-state computation, with obvious benefits for code size. Both methods enable significantly larger DFAs, but on an IXP-like NPU they require more memory reads, which lowers the performance. *Content Addressed Delayed Input DFA* [17] solve this issue by making the number of memory reads when following *default* transitions equal to an uncompressed DFA. In Ruler the emphasis is on the rewriting functionality, so we did not implement any of the advanced implementation techniques for DFAs.

A modification of the classic Aho-Corasick pattern-matching algorithm proposed in [9] reduces the number of reads from the input by consuming multiple bytes at once. This approach requires parallel execution of the same DFA on the same input with different offsets. Combining this method with bit-level parallelism to decrease the fan-out of the transitions achieves throughput of 5 Gbps [20] on a fast IXP2855. Each thread runs one instance of the DFA. Using the same extrapolation on the older and less powerful IXP2400, the performance is comparable with our interpreted states in memory. Beside rewriting, our contribution is that if the execution hits the interpreted states rarely, the overall performance of our hybrid implementation is higher.

7. CONCLUSIONS AND FUTURE WORK

We have developed Ruler, a programming language and compiler that can generate efficient code for a wide range of processor architectures and is especially helpful on NPUs. With Ruler, the programmer does not have to worry about details of the processor architecture: they are all handled by the compiler. Moreover, Ruler software is portable, because the compiler can target a wide range of processing platforms, ranging from FPGAs to user- and kernel-space programs on standard processors. Finally, since Ruler does not only offer high-speed pattern matching operations, but also rewriting, it offers significantly new functionality compared to existing pattern-matching tools for network processing.

In this paper we have shown our implementation for the IXP series of NPUs, including benchmarks for a wide range of packet matching and rewriting operations. Considering that the IXP2400 that we used in our experiments is nowadays a cheap commodity component, the results are encouraging. Ruler makes the NPU easily programmable, and it can handle significant pattern matching and rewriting tasks at Gigabit link rate.

Targeting a different processor in the IXP series, such as the IXP28xx and IXP23xx is almost directly possible, and we expect that retargeting a different series of multi-core chips would be straightforward too.

Another future extension of the Ruler is processing of whole streams. The language itself has all the necessary features to classify TCP connections and the Ruler engines can restart execution of the DFA when it receives the next packet of a stream. However, the current challenge is to handle missing, retransmitted and out of order TCP packets.

Ruler takes full advantage of the multi-core IXP2xxx chip. With the upcoming multi- and many-core general purpose chips (e.g., Niagara2), Ruler can be easily used for offloading the monitoring tasks from the operating system to the bare-metal cores. In addition, the TCP stack within the operating system can prepare the TCP streams for deep inspection.

Acknowledgments

We would like to thank Lennert Buytenhek for his invaluable help during development of the IXP code and Willem de Bruin for commenting on earlier versions of this paper. This research was funded by the EU FP6 Lobster project and supported by University of Otago and World45 Ltd., New Zealand.

8. REFERENCES

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[2] Z. K. Baker and V. K. Prasanna. High-throughput linked-pattern matching for intrusion detection systems. In *ANCS '05: Proceedings of the 2005 symposium on Architecture for networking and communications systems*, pages 193–202, New York, NY, USA, 2005. ACM Press.

[3] H. Bos and K. Huang. Towards software-based signature detection for intrusion prevention on the network card. In *Proceedings of Eighth International Symposium on Recent Advances in Intrusion Detection (RAID2005)*, Seattle, WA, September 2005.

[4] B. C. Brodie, D. E. Taylor, and R. K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 191–202, Washington, DC, USA, 2006. IEEE Computer Society.

[5] S. Carr and P. Sweany. Automatic data partitioning for the agere payload plus network processor. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 238–247, New York, NY, USA, 2004. ACM Press.

[6] R. K. Cunningham, R. P. Lippmann, D. J. Fried, S. L. Garfinkel, I. Graf, K. R. Kendall, S. E. Webster, D. Wyschogrod, and M. A. Zissman. Evaluating intrusion detection systems without attacking your friends: The 1998 DARPA intrusion detection evaluation. In *SANS 1999*, 1999.

[7] 1998 DARPA intrusion detection evaluation data set. webpage. `www.ll.mit.edu/IST/ideval/data/1998/1998_data_index.html`.

[8] W. de Bruijn, A. Slowinska, K. van Reeuwijk, T. Hruby, L. Xu, and H. Bos. Safecard: a gigabit ips on the network card. In *Proceedings of 9th International Symposium on Recent Advances in Intrusion Detection (RAID'06)*, Hamburg, Germany, September 2006.

[9] S. Dharmapurikar and J. Lockwood. Fast and scalable pattern matching for content filtering. In *ANCS '05: Proceedings of the 2005 symposium on Architecture for networking and communications systems*, pages 183–192, New York, NY, USA, 2005. ACM Press.

[10] U. Erlingsson, M. S. Krishnamoorthy, and T. V. Raman. Efficient multiway radix search trees. *Information Processing Letters*, 60(3):115–120, 1996.

[11] J. Gait. Hash table methods for case statements. In *ACM-SE 20: Proceedings of the 20th annual Southeast regional conference*, pages 211–216, New York, NY, USA, 1982. ACM Press.

[12] G. Gilder. Telecosm: how infinite bandwidth will revolutionise our world. *Audio-Teck Business Book Summaries*, 9(12):9–24, Dec. 2000.

[13] Intel. Intel©IXP2XXX Product Line of Network Processors. http://www.intel.com/design/network/products/npfamily/ixp2xxx.htm.

[14] Intel. Intel©XScale Technology. http://www.intel.com/design/intelxscale/.

[15] S. J., O. R., and A. Kuznetsov. Beyond Softnet. In *5th Annual Linux Showcase and Conference*, pages 165–172, November 2001.

[16] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 339–350, New York, NY, USA, 2006. ACM Press.

[17] S. Kumar, J. Turner, and J. Williams. Advanced algorithms for fast and scalable deep packet inspection. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 81–92, New York, NY, USA, 2006. ACM Press.

[18] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[19] V. Laurikari. NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. *SPIRE*, 00:181, 2000.

[20] P. Piyachon and Y. Luo. Efficient memory utilization on network processors for deep packet inspection. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 71–80, New York, NY, USA, 2006. ACM Press.

[21] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th Conference on Systems Administration (LISA-99)*, pages 229–238. USENIX, November 1999.

[22] J. Wagner and R. Leupers. C compiler design for an industrial network processor. In *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 155–164, New York, NY, USA, 2001. ACM Press.

[23] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 93–102, New York, NY, USA, 2006. ACM Press.

[24] F. Yu, R. H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using tcam. In *ICNP '04: Proceedings of the Network Protocols, 12th IEEE International Conference on (ICNP'04)*, pages 174–183, Washington, DC, USA, 2004. IEEE Computer Society.