

Enabling Flexible Packet Filtering Through Dynamic Code Generation

Original

Enabling Flexible Packet Filtering Through Dynamic Code Generation / Morandi, Olivier; Risso, FULVIO GIOVANNI OTTAVIO; Baldi, Mario; Baldini, A.. - (2008), pp. 5849-5856. (IEEE International Conference on Communications, 2008. ICC '08. Beijing (China) 19-23 May 2008) [10.1109/ICC.2008.1094].

Availability:

This version is available at: 11583/1667111 since:

Publisher:

Published

DOI:10.1109/ICC.2008.1094

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

Enabling Flexible Packet Filtering Through Dynamic Code Generation

O. Morandi, F. Risso, M. Baldi
Dipartimento di Automatica e Informatica
Politecnico di Torino
Torino, Italy

A. Baldini
Cisco Systems
San Jose, CA, USA

Abstract— Despite its efficiency, the general approach of hardcoding protocol format descriptions in packet processing applications suffers from many limitations. Among the others, the lack of flexibility when needing to extend the software for supporting new protocols, and the proliferation of modules with similar functionality between different applications, resulting in decreased maintainability. The NetPDL language was defined for overcoming such limitations, allowing decoupling applications from the knowledge of the format of protocol headers. The main criticism to NetPDL relates to its supposed performance penalties; this paper demonstrates that this language can be effectively used for the dynamic generation of optimized, i.e. efficient and fast, packet-processing code, and presents the architecture of a compiler implemented for such purpose.

I. INTRODUCTION

Packet processing applications such as routers, firewalls and IDSs heavily rely on some routines that locate or get the content of some specific fields in network packets. Traditionally, these modules follow the traditional approach of hardcoding the format of protocol headers in the software, which is no longer a viable solution because of non-negligible limitations in terms of flexibility and maintainability. Developers must have a deep knowledge of protocol header formats and adding support for new protocols implies modifying the application, debugging and testing it again. Besides, different applications that rely on similar protocol decoding functionalities are usually based on custom code, which results in a multiplication of the amount of software to be written and maintained, with a corresponding increase in the incidence of bugs and security flaws.

The Network Protocol Description Language (*NetPDL*) has been defined for overcoming such limitations and aims at describing the format of network protocol headers and encapsulation rules between different protocols. Using NetPDL, a packet processing application does not need any direct knowledge on how to locate header fields inside a packet buffer, since such information is provided by an external protocol description database. In [2] the authors show how NetPDL can be profitably used for implementing a packet-decoder, i.e. an engine for parsing the content of network packets and for extracting the actual values of each field, through a step-by-step interpretation of an external protocol description database. Such module is now part of the NetBee library [3] and it is used for visualizing packet-data in the Analyzer [4] network sniffer.

Although these first experiments prove the feasibility and the potential of NetPDL-based applications, the measured performances are not compatible with the requirements of high speed data-plane applications, so the main criticisms to NetPDL are still focused on its supposed performance penalties with respect to solutions based on hardcoded protocol descriptions. We object that performances are not related to the language itself, but to the tools using it. In fact, in order to take full advantage of a solution based on NetPDL, protocol descriptions should be translated to a native language through a compiler, thus eliminating the overheads caused by interpretation and enabling the deployment of any suitable optimization on the generated code.

This paper presents the architecture of a compiler for the translation of NetPDL-based packet filtering rules into binary code for the Network Virtual Machine (*NetVM*) [5] [6] demonstrating that NetPDL protocol descriptions can be effectively used for driving the dynamic generation of specialized packet processing programs. In our solution, the code generation process is decoupled from the knowledge of the format of protocol headers, which resides in an external NetPDL database. In particular, NetPDL protocol descriptions are translated into programs that implement high level filtering rules expressed in the Network Packet Filtering Language (*NetPFL*).

This work focuses on the implementation of a compiler for packet filters; hence it does not intend to propose a new packet filtering model in competition with other well known solutions such as the BPF [7], or the more recent FPF [8]. Indeed, we argue that packet filtering functionalities provide the basic blocks for building complete protocol-agnostic applications because they can be easily extended in order to support more complex operations, like field extraction and more.

This paper is structured as follows. Section II provides an overview on the basic building blocks that represents the foundation of this work. An algorithm for the dynamic generation of packet filtering programs from NetPDL protocol descriptions is presented in Section III, while Section IV discusses its implementation in a compiler. The performances of the generated code are assessed in Section V, and conclusions are drawn in Section VI.

II. RELATED TECHNOLOGIES: NETPDL, NETPFL AND NETVM

The compiler presented in this paper fits into a more complex architecture (shown in Figure 1), in which the most important pieces are the NetPFL filtering language, the NetPDL language, and the NetVM virtual machine. The NetPFL represents the language used to define packet filters, while the NetPDL database contains the descriptions of the supported protocols. Our compiler takes the packet filter and, according to the protocol format and encapsulation rules, generates a piece of assembly code that is executed by the NetVM virtual machine, which resides on the “data path”. For the sake of precision, in performance-sensitive environments the NetVM assembly is further translated into a native assembly in order to create a program targeted to the real hardware platform (e.g., a network processor). However, this is outside the scope of this paper. In the rest of this section we give an overview on these building blocks, while the following sections will focus on the compiler architecture, and on the deployed code generation techniques.

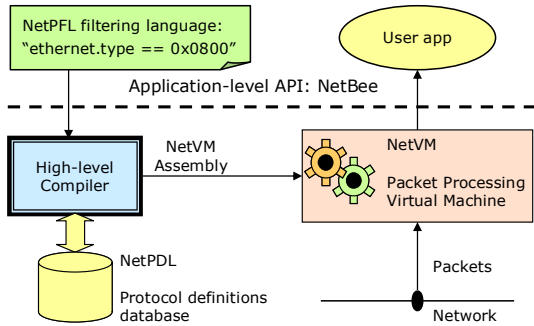


Figure 1. Complete view of the proposed packet processing architecture.

A. NetPDL

The NetPDL language (the complete specification is available in [1]) enables the description of how protocol headers are laid out and chained together inside network packets. Since it is based on XML, the elements of the language are identified by specific tags, each one characterized by several attributes and organized in hierarchical structures.

Describing a protocol in NetPDL means enclosing in a section identified by the `<protocol>` tag the list and the binary format of the fields that build up a header, as well as the encapsulation relationships that can be present between different protocols. Figure 2 shows a sample NetPDL specification for the Ethernet protocol header. In the `<format>` section we find the description of the binary layout of the header as a list of `<field>` elements. The `<encapsulation>` section, on its side, identifies the conditions that need to be met for other protocols to be encapsulated into the one being described. For instance, the `<nextproto>` element, acts as a pointer to the next protocol header.

NetPDL allows the description of complex headers through the definition of several kinds of header fields (e.g., fixed and variable size fields, bitfields, padding and more) and through

the use of structured control flow constructs, such as *if-then-else*, *switch-case*, and *loop*. Conditional elements can appear also in the `<encapsulation>` section for describing complex encapsulation rules.

```
<protocol name="ethernet" longname="Ethernet 802.3">
  <format>
    <fields>
      <field type="fixed" name="dst" size="6"/>
      <field type="fixed" name="src" size="6"/>
      <field type="fixed" name="type" size="2"/>
    </fields>
  </format>
  <encapsulation>
    <switch expr="buf2int(type)">
      <case value="0x800"> <nextproto proto="#ip"/> </case>
      <case value="0x86DD"> <nextproto proto="#ipv6"/> </case>
    </switch>
  </encapsulation>
</protocol>
```

Figure 2. NetPDL description of the Ethernet protocol header.

The language specification also includes the definition of two fictitious protocols that are named `startproto` and `defaultproto` with the purpose to provide respectively the entry and the exit points to the protocol encapsulation.

B. Packet Filtering and Fields Extraction: NetPFL

Even though the NetPDL provides features that go beyond those of a completely declarative language, its only purpose is the description of the format of network protocol headers and it provides no means for defining actions to be executed when specific conditions are satisfied. A simple classification language called *Network Packet Filtering Language (NetPFL)* has been defined to provide such features.

NetPFL is based on a filter-action model to express packet filtering conditions and packet handling statements, such as accepting a packet or extracting the actual values of a set of fields. Filters can be based on (i) protocols (i.e. a filter is satisfied if the packet contains the specified protocol header), and (ii) field values (i.e. a filter can be specified as an expression involving the value of one or more header fields). Basic predicates can be composed with the Boolean operators AND, OR, and NOT in order to express complex filters. Figure 3 shows two sample NetPFL rules: the first represents a filtering condition on the `ip.src` field, while the second is a field extraction statement for returning the values of the `ip.src` and `ip.dst` fields contained in each `ip` packet.

```
ip.src == 10.0.0.1 returnpacket
ip extractfields(ip.src, ip.dst)
```

Figure 3. NetPFL expression examples.

NetPFL is built on top of NetPDL as its main tokens (i.e. protocol names and header fields) are not specified explicitly, but are defined in a NetPDL database. In other words, the filter expressed in Figure 3 makes sense only if the NetPDL description contains the definition of a protocol named “ip” whose header contains a field named “src”. This characteristic makes our compiler definitively more complex because it must be able to work with changing tokens without being recompiled.

C. NetVM

Our target for dynamic code generation is represented by the Network Virtual machine (*NetVM*) [6], an abstract packet-handling engine that allows the portability of network processing applications across heterogeneous architectures.

In NetVM a packet-processing program is expressed as a set of modules called *Network Processing Elements (NetPEs)*, which represent virtual processors that execute a mid-level assembly language called *Networking Intermediate Language (NetIL)*. The interconnections between different modules determine the behavior of the entire application. The elementary execution engine, the NetPE, is a stack-based processor (hence the NetIL is a stack-based language) that is made up of a set of private registers (e.g. stack pointer, etc.) and a hierarchy of memories, such as a local memory for storing state information that is local to a processing engine, and an exchange memory for storing the packet buffer and additional metadata.

As for Java applications, the execution of a NetVM program on real hardware relies on the existence of an implementation of the virtual machine, which can be an interpreter or a compiler for the translation of NetIL code to native machine code. Figure 4 shows an example on how a simple packet-forwarding element can be implemented as a NetVM application.

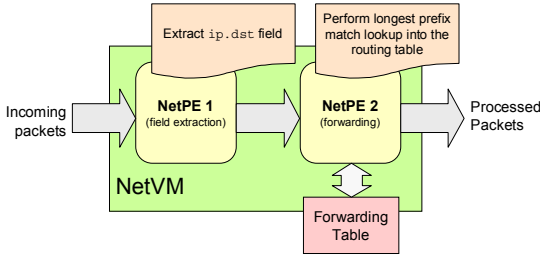


Figure 4. NetVM Based Forwarding Element.

III. GENERATING PACKET PROCESSING CODE FROM NETPDL

In our compiler, we consider a packet filter as a program composed by two main sections: (i) a packet demultiplexing section, where the sequence of the headers carried by each packet is analyzed looking for a specific protocol, and (ii) a section where some conditions on one or more fields are evaluated and the corresponding action is triggered. In other words, the packet filter looks for the first occurrence of the specified header inside the packet and then checks some conditions on one or more of its fields, as shown in Figure 5. In our discussion we will focus mainly on packet filtering, because field extraction programs follow a scheme that is very similar to the one described, except that field values are loaded from the packet buffer and used by other modules instead of being evaluated by filtering conditions.

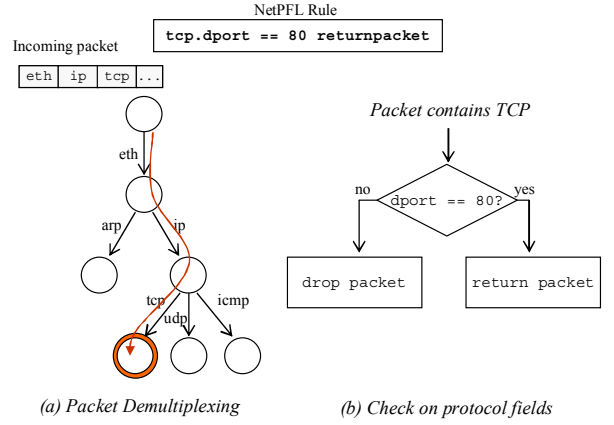


Figure 5. Filtering program are the composition of (a) a packet demultiplexing section and (b) a section for checking conditions on protocol fields.

A. The Protocol Encapsulation Graph

Considering a NetPDL database, encapsulation relationships that exist between protocols can be used to identify a directed graph $G(V,A)$, where each node V represents a protocol in the database, and an edge $e(x,y)$ is directed from the node x to the node y , if the protocol y can be encapsulated into the protocol x . We call such a graph a *Protocol Encapsulation Graph*, or *encapsulation graph*.

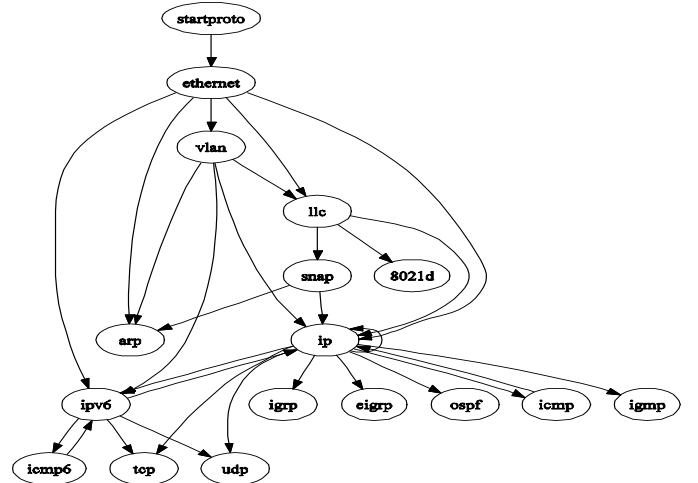


Figure 6. Protocol Encapsulation Graph.

The encapsulation graph exposes the layered nature of network protocols and has some similarities with the concept of *Protocol Graph*, i.e. a directed acyclic graph employed for describing the *use* relations existing between the different components of a multi-protocol communications system [7]. However the encapsulation graph allows paths between nodes to be cyclic, making evident the cases of protocols that can be tunneled, like IPv4 encapsulated in IPv4, IPv6 in IPv4 and vice-versa, or cases like an ICMP message encapsulated in IPv4, which carries a further IPv4 header (belonging to the packet that generated the message), and more.

Figure 6 shows how complex an encapsulation graph can be. In particular, it shows the encapsulation graph

corresponding to a subset of the current NetPDL database, containing only some protocols up to the transport layer.

B. Packet Demultiplexing

In our model, the first section of a generic packet filter needs to parse the sequence of headers, while looking for a specific protocol. Since the encapsulation graph represents the union of all the demultiplexing paths that lead to every protocol defined in a NetPDL database, we can leverage such information by considering only the set of paths that lead to the protocol we are looking for, i.e. a sub-graph of the encapsulation graph. Since the characteristics of the encapsulation graph ensure that a single source node always exists (i.e. the node corresponding to the `startproto` protocol), a *reverse postorder*¹ visit starting from a generic node N will identify a subgraph that is the union of all the paths leaving from the `startproto` node, leading to N itself.

```

Procedure GenFilterCode(Node n, Expr e)
  Begin
    TargetProtocolNode = n
    For each p in EncapsulationGraph
      p.visited = false

    RPO_Visit(n)
    If (e)
      GenCodeForSection(TargetProtocolNode.Format)
      GenCodeForExpr(e)
    If (!TargetProtocolNode.successors.empty())
      GenCodeForSection(TargetProtocolNode.Encapsulation)
    End

Procedure RPO_Visit(Node n)
  Begin
    If (n.visited)
      Return
    n.visited = true
    For each p in n.predecessors
      RPO_Visit(p)
    GenCode(n)
  End

Procedure GenCode(Node n)
  Begin
    If (n ≠ TargetProtocolNode)
      GenCodeForSection(n.Format)
      GenCodeForSection(n.Encapsulation)
  End

```

Figure 7. Code Generation Algorithm.

Given such considerations, our strategy for generating a packet filtering program through NetPDL is presented in the algorithm of Figure 7. The code generation process is driven by the `GenFilterCode()` procedure that accepts as arguments the node corresponding to the protocol on which the source filter is set (e.g. “ip”), and an optional expression evaluating some of its fields (e.g. “dst == 10.0.0.1”). Briefly, the algorithm performs a reverse postorder visit on the encapsulation graph starting from the target node (i.e. the node relative to the protocol to be searched). Then, it generates the code related to the format (which is required in order to be able to locate every field of the selected protocol) and the encapsulation (which is required to be able to link the current

¹ A *reverse postorder traversal* of a directed graph is a *dept-first search* visit, in which every node is visited after all its predecessors. For instance, if the selected protocol is “llc” the reverse postorder visit of the graph in Figure 6 will be: startproto, ethernet, vlan, llc.

protocol to its successor nodes) sections, for all the protocols encountered during the visit. In particular, the encapsulation section can be modelled as a multi-target branch instruction, i.e. a generic *switch-case* construct, which evaluates the content of some header fields, and where each branch leads to the code generated for the protocols corresponding to the successor nodes of the one being visited, while a special branch is directed to a “filter-false” exit label for indicating the absence of a match. Some exceptions arise for the target protocol (i.e., the protocol we want to locate), in which the code has to be generated in a slightly different manner. For example, if the source filtering expression evaluates some fields of the target protocol header, the `GenCodeForSection()` procedure is invoked in order to generate a portion of code for locating them, while the `GenCodeForExpr()` generates the final check. Furthermore, if the target protocol node has any successors (the encapsulation graph can contain loop) the `GenCodeForSection()` procedure translates its encapsulation section, giving the opportunity to find a match in subsequent tunneled instances of the same protocol header, even if the current header does not match the filter. For instance, in case of an IPv4 in IPv4 tunneling the external IP header may not match the filter, while the internal one can.

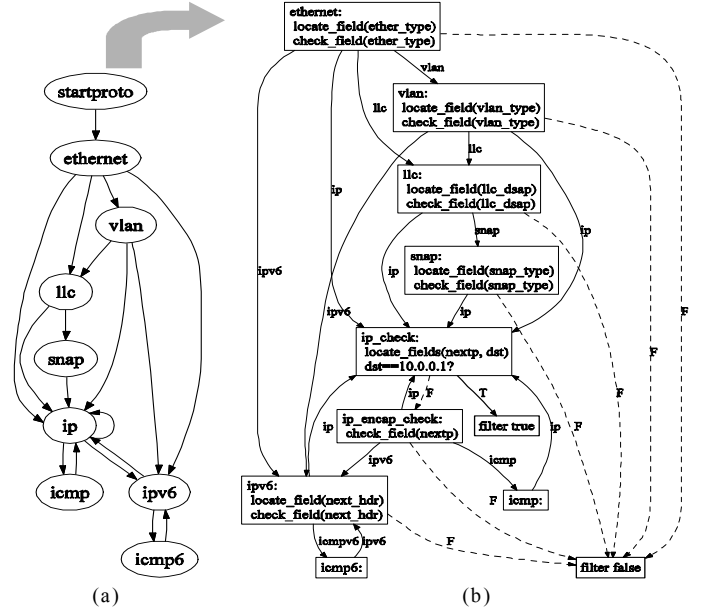


Figure 8. (a) Demultiplexing Paths and (b) Control Flow Graph for the filter “ip.dst == 10.0.0.1 returnpacket”

Figure 8 shows the results of the two phases of the code generation process for the NetPFL rule defined in the example: (a) shows the portion of the encapsulation graph representing all the demultiplexing paths that lead to IP, while (b) shows the representation of the generated code as a control flow graph.

The sample filter is matched when the first IP header containing a destination address field equal to the 10.0.0.1 is found. If the first IP header does not match the filtering condition, the program continues to parse the packet by following the demultiplexing paths of the subgraph until it

finds a match, or it reaches a terminal node (e.g., the end of the packet).

C. Locating header fields

In NetPDL, every field declaration not only identifies a specific sequence of bytes into the packet buffer, but implicitly tells where the next field will start. In particular, the offset of a header field defined in a NetPDL database is not specified explicitly, but it can be implicitly derived by adding the offset and the size of its preceding field, as in (1).

$$\text{Offs}(\text{Field}_i) = \text{Offs}(\text{Field}_{i-1}) + \text{Size}(\text{Field}_{i-1}) \quad (1)$$

This rule can be used to map the protocol format into a sequence of instructions for identifying the actual offset and size of every field. Unfortunately, most protocols include fields whose size is known only at run-time, which prevents this computation to be performed at compile-time. Besides, since different packets can take different demultiplexing paths, even the starting offset of a specific header cannot be known in advance. Given such considerations, the cleanest way for generating a portion of code for locating header fields inside packets is to translate the entire `<format>` section of a NetPDL description to a sequence of instructions that implement the scheme described in (1), and to delegate the task of removing useless and redundant code to a series of optimization steps. Such choice is based on the fact that the evaluation of the content of some fields performed in encapsulation and filtering conditions can be treated like *uses* of particular *variables* (i.e. the fields). Using simple data-flow analyses, the instructions defining variables that will never be used can be detected and safely removed. Moreover, the definitions of fields of fixed size can be subject to the application of *constant propagation* techniques. Section IV.B will provide more details on such topic.

IV. THE COMPILATION PROCESS

We implemented the techniques described in the previous section in a compiler for the translation of NetPFL rules into executable code for the NetVM virtual machine, through the exploitation of the information on the format of network protocols resident in an external NetPDL database. The compiler adopts a traditional architecture that includes a front-end component that translates the source program in a more manageable intermediate representation (IR), an optimizer, and a back-end for the generation of the target executable code.

A. Code Generation

In a first phase the compiler parses the NetPDL protocol database by gathering the names of protocols and fields. At the same time the encapsulation graph is created for modelling the encapsulation information defined in the NetPDL description. Then the source NetPFL rule is parsed, while ensuring that the filtering expression refers to available protocols and fields. If the filtering expression is made up of terms related to different protocols, the parser also tries to group together sub-expressions that include terms referring to the same protocol. This ensures that each one of such sub-expressions can be

implemented by (i) a demultiplexing program for searching the specified protocol and (ii) a portion of code for checking the values of fields of the header. In such way, a compound filter (i.e., which refers to different protocols) can be generated through the algorithm reported in Figure 7 for each sub-expression referring to the same protocol, and by linking together all such portions of the program, as shown in Figure 9. The optimization of composed filters is left to future work.

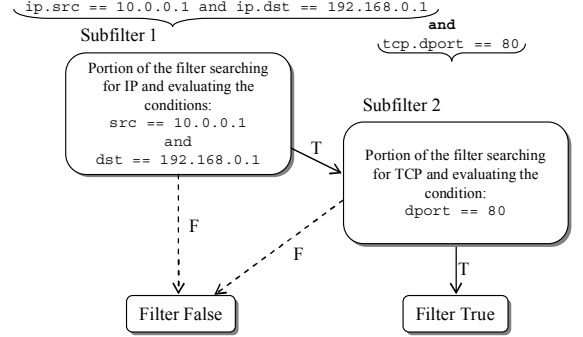


Figure 9. Composed filter.

During the IR generation phase, all the encapsulation and filtering conditions referring to fields are translated into checks on integer values loaded from the packet memory (if the size of the field is less than or equal to 4 bytes), or into string comparison operations (for fields greater than 4 bytes). References to bit-fields are translated into masking operations on values loaded from the packet buffer. Finally, structured control flow constructs such as *if-then-else*, and *loops* are lowered to explicit branch operations.

The generated intermediate representation of the resulting filtering program can then be optimized and finally translated to the target NetVM executable code.

B. Optimizations

The translation of NetPDL descriptions into sequences of instructions for locating header fields produces a large amount of redundant code, which is reduced through a set of optimization steps. In particular, the definitions of variables that are never used are identified and safely removed by a *dead store elimination* phase, while a *constant propagation* phase recognizes the variables that hold a constant value and substitutes their use with the direct use of the constant. Since constant propagation can transform expressions evaluating variables in expressions evaluating only constant values, it is supported by a *constant folding* phase for substituting such sub-expressions with their result computed at compile-time. Besides, the lowering to explicit branch instructions of structured control flow constructs produces several sequences of jump to jump instructions that can be easily individuated and coalesced by inspecting the control flow graph.

The quality of the generated code could be further improved by applying more specialized optimizations like those proposed by Begel et. al. in [11] for eliminating redundant checks on the same fields and for reducing the overall depth of the control flow graph of composed filters;

however the implementation of such algorithms was outside the scope of our current work.

C. Considerations on Safety

Filtering programs produced by our compiler are supposed to be executed in a safe virtual machine environment, where unbounded memory accesses are disallowed and backward pointing branch instructions are strongly limited to the cases where branching conditions can be evaluated to be constant or bounded at bytecode-load time, for ensuring that the program terminates in a finite time. The former point implies that packet memory bounds checks can be delegated to the NetVM runtime environment, where accesses outside the limits of the packet will raise an exception and will make the filter to fail. The latter point indeed has important implications on the translation of complex protocol descriptions like the one of IPv6, which contains an uncontrolled *while-do* loop for decoding the extension headers. In our compiler, this problem is addressed by defining an upper bound to the number of cycles, which can be specified at compile time. In this way the IPv6 protocol is supported, although with a limit on the consecutive extension headers allowed in IPv6 packets.

Similar considerations arise when considering the loops generated by tunnelled encapsulations (e.g. IP – GRE – IP), although in this case the solution is more complex because a preventive analysis of the encapsulation graph should be performed for determining the protocols involved in cyclic paths, and a mechanism for limiting the number of such loops should be put in place into the generated code.

A detailed analysis on safety issues and loop bounding is reserved to a future work.

V. EVALUATION AND RESULTS

This section assesses the ability of our compiler to generate NetIL filtering programs from simple NetPFL rules and compares the results with equivalent filters generated for the BPF virtual machine by the well-known `libpcap/tcpdump` tools and with native filters written in C and compiled with a general-purpose C compiler. As an example, translating the NetPFL rule

```
ip.dst == 10.0.0.1 returnpacket
```

into executable code for the NetVM virtual machine results in the optimized filtering program shown in Figure 10². The corresponding BPF filter generated through the `tcpdump` tool is shown in Figure 11. Besides the intrinsic differences between BPF and NetVM architectures (i.e. the NetVM is stack-based while the BPF virtual machine is register based), we can see that two programs are functionally equivalent. Both check the Ethernet `type` field against value `0x800`, then check if the IP destination field contains address `10.0.0.1`; the packet is accepted only if both conditions are true. The primary difference between the two approaches is not immediately visible, because it relates to the simplicity in adding support for new protocols (e.g. a new data-link layer protocol). In the case

² This example uses a limited NetPDL database including only Ethernet and IP for the sake of clarity.

of the presented compiler it is sufficient to update the XML file containing NetPDL protocol descriptions, while in the other case some of the `libpcap` source files must be modified and the library must be recompiled.

```
push 12          ;offset of the ethertype field
upload.16        ;load the ethertype field
push 2048        ;0x800
jcmp.neq DISCARD ;compare and jump to DISCARD if not equal
push 30          ;offset of the ipdst field
upload.32        ;load the ipdst field
push 167772161   ;10.0.0.1
jcmp.neq DISCARD ;compare and jump to DISCARD if not equal

ACCEPT:
  pkt.send out1   ;filter true

DISCARD:
  ret             ;filter false
```

Figure 10. NetIL code generated from the NetPFL rule “ip.dst == 10.0.0.1 returnpacket”.

```
(0) ldh [12]          ;load the ethertype field
(1) jeq #0x800      jt 2 jf 5 ;if ==0x800 goto 2, else goto 3
(2) ld [30]         ;load the ipdst field
(3) jeq #0xa000001 jt 4 jf 5 ;if ==10.0.0.1 goto 4, else goto 5
(4) ret #1514       ;return the frame length
(5) ret #0          ;return false
```

Figure 11. BPF code for the `libpcap` filter “ip dst 10.0.0.1”.

In order to evaluate the performances of the filtering programs produced by our compiler we profiled the execution of five simple (and common) filters³ generated using both a reduced NetPDL database and a complete one. While the first number is used to compare the NetPDL-based technology with equivalent BPF and natively programmed filters (which support a limited number of encapsulations), the second set of test is used to show the flexibility of our approach, and to demonstrate that the completeness of the NetPDL database does not affect performance on common network traffic. The NetPDL database used was the one available online at the time of writing (February 2008), which includes 122 protocols and whose size is 993KB; since the generated code depends on the database, results obtained with a different version may vary.

All the tests were performed on a 3 GHz Intel Pentium 4 machine with Hyper-threading and 4GB of memory. NetVM and BPF programs were compiled Just in Time into x86 assembly and executed in user space⁴. Native filters were programmed in C language and compiled through the Microsoft Visual C++ 2005 compiler. Each test was executed by applying the filtering on a packet created on purpose, in a way that all the conditions of the filter had to be checked before returning the result. Measurements were done through the RDTSC instruction available on Intel CPUs, and special care has been done in order to prevent problems due to variable clock speed, hyperthreading and instruction reordering. Results

³ Filter, according to the NetPFL syntax are “ip” (filter1), “ip.src == 10.1.1.1” (filter2), “tcp” (filter3), “ip.src == 10.1.1.1 and ip.dst == 10.2.2.2 and tcp.sport==20 and tcp.dport == 30” (filter4) and “ip.src == 10.4.4.4 or ip.src == 10.3.3.3 or ip.src==10.2.2.2 or ip.src == 10.1.1.1” (filter5). WinPcap syntax is equivalent, although is not reported here for the sake of brevity.

⁴ BPF programs have been compiled to native code using the Just in Time compiler provided by the WinPcap library, which is an implementation of the `libpcap` library for Microsoft Windows.

are related to average execution time of each filtering program, excluding all other overheads (e.g. function call, RDTSC cost).

Figure 12 shows the time required to execute the abovementioned filters by interpreting BPF and NetVM assembly programs, generated by using both a reduced and a full NetPDL database. Although this first set of results does not appear so encouraging (BPF filters outperform NetPDL-based ones several times) the reason is mainly due to the differences between the architectures of the BPF and the NetVM. Indeed, the necessity of emulating the NetVM operand stack implies a major overhead over the BPF, which is register based.

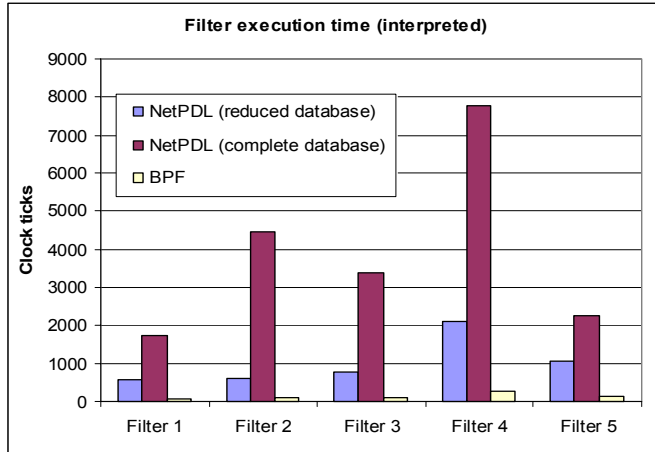


Figure 12. Average processing times for five different filters, interpreted.

A more appropriate set of results is shown in Figure 13, which compares the same filters translated into native assembly. A first result is that our compiler infrastructure performs better than the JIT compiler available in WinPcap. The reason is that our compiler includes several optimization steps, while the JIT implemented in WinPcap is basically a translator of BPF instructions into x86 assembly, without optimizations. This result demonstrates our claim that the NetPDL does not insert performance penalties and that the results depend only on the quality of the tools using it NetPDL-based tools.

The second result shown in Figure 13 is that filtering programs generated from a reduced NetPDL database and compiled into native x86 code by our framework have performances that are better than the ones of native filters programmed in C language compiled using a full-featured commercial compiler, which provides no flexibility at all. Notably, this result has been obtained with our NetVM JIT compiler that implements only a basic set of optimizations, compared to the more aggressive optimization techniques implemented in a commercial C compiler. One of the reasons is that on little-endian architectures, such as the Intel IA32 processors, data larger than one byte that is read from the packet buffer must be converted from big-endian (the network byte order) to little-endian (the host byte order of x86 processors). In native filters such operation is performed through the library functions of the `ntoh()` family, while the code generated by the NetVM JIT compiler directly uses the `BSWAP` (byte order swap) x86 instruction that is far more efficient. Our results are usually still better even in case `ntoh()`

functions are replaced with an ad-hoc macro in native filters (second column in Figure 13), because the NetVM model facilitates the implementation of more network-oriented optimizations in the code, even if the quality of our compiler is far from reaching the one of other tools such as Microsoft Visual Studio or GNU GCC.

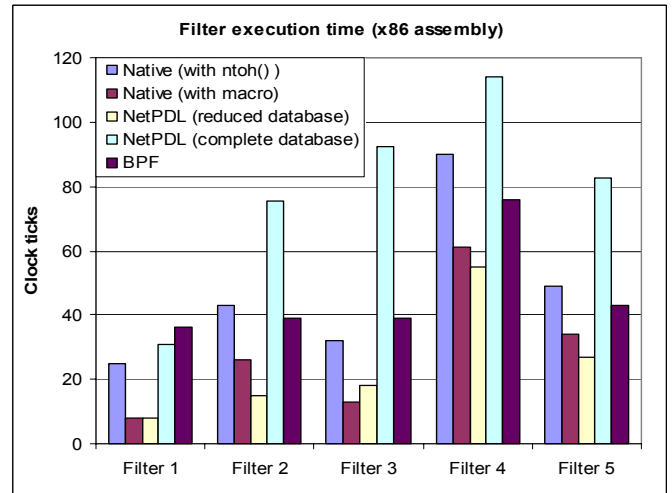


Figure 13. Average processing times for five different filters in case of native assembly.

An unexpected (but very important) result is that, according to our tests, the pure C language may not be the perfect choice for writing efficient packet processing code. Although our example does not have general validity, the C language does not have the notion of packets, hence any data is a plain buffer. This requires for example the use of functions that take into account byte ordering issues (e.g., `ntoh()`) when accessing to protocol data; however a general-purpose compiler finds a hard way to optimize these functions. A special purpose language, coupled with a dedicated set of tools for code generation, can solve the same issue much more efficiently, ranging from using dedicated assembly instructions (e.g. the `bswap` opcode), to the use of pre-processed data. Particularly, this technique offers valuable speed-ups because it is based on the semantic of data; for example, it is able to translate immediately a user input (e.g. an IP address) into the “native format” (i.e., network byte order), and use the new value when checking the content of a field in the packet without any performance penalties. However, a more in-depth investigation of these issues is left for future work.

Since NetPDL supports a wide variety of protocols and cyclic encapsulations, as Figure 6 shows, the programs produced by our compiler are way larger than the corresponding BPF filters. For instance a non-optimized IP filter generated using the standard NetPDL database counts 292 statements, versus 4 statements of the corresponding BPF program, as show in Table 1. However, while BPF and native programs only identify IP packets directly encapsulated within a lower layer packet, the abovementioned NetPDL-derived program identifies IP packets encapsulated in several possible ways (e.g., an IPv4 packet tunnelled within another IPv6 packet). It should be noted that the higher number of instructions generated by the compiler does not correspond to

the number of instructions effectively executed in the “fast path” of the code (i.e. the typical number of instructions executed at runtime on common packet traces), however as Figure 13 shows, the capability of recognizing complex encapsulations comes at a cost in terms of performances, because all the possible cases must be taken into account.

TABLE 1. NUMBER OF STATEMENTS GENERATED BY DIFFERENT COMPILERS.

	Filter1	Filter2	Filter3	Filter4	Filter5
BPF interpreted	4	6	6	17	9
NetIL interpreted (reduced database)	10	14	23	76	26
NetIL interpreted (complete db)	292	491	487	1544	497
C-hardcoded filters, x86	14	29	23	78	41
BPF x86	43	61	59	170	70
NetIL x86 (reduced db)	14	20	25	77	33
NetIL x86 (complete db)	494	834	1348	3557	844

Currently, the NetPFL compiler is not optimized for speed in code generation. For instance, the libpcap compiler needs about 120 μ s to compile the “tcp.dport == 80” filter, against 87ms of the NetPFL. Although this value is still reasonable, this result is mostly due to the very different number of statements generated by the two compilers before optimizations, which differs of about two orders of magnitude, as shown in Table 1 (first and third lines). It is worth recalling that the compilation time usually grows non-linearly with program size. For completeness, the number of instructions generated by the several compilers involved in our tests (BPF interpreted and JIT compiled, native filters with the ntohs()-equivalent macro, ad NetIL interpreted and JIT compiled with both the reduced and complete NetPDL database) are reported in Table 1.

VI. CONCLUSIONS

This paper demonstrates that the NetPDL language does not insert performance penalties when developing packet-processing applications. This result enables a novel approach to the development of such these applications, based on decoupling the application logic from the knowledge about the format of network protocols, which resides in an external NetPDL protocol description database. A compiler for packet filters has been developed following such approach, which demonstrates that the dynamic generation of efficient filtering

programs from NetPDL is feasible and can lead to performance comparable to the one of equivalent C language programs, with the advantage of adding support for new protocols or new encapsulation paths without changing the application code.

The presented solution, although powerful, has the limitation of supporting protocols only up to the transport layer. However, NetPDL has recently been extended with features for application layer protocol classification and recognition. Hence, future efforts will be directed towards the integration of such features in the presented compiler and NetPFL. Besides, the structure of the Protocol Encapsulation Graph should be investigated in more detail, since we believe that it can be exploited to optimize the generation of code for composed filters by merging the sub-graphs relative to each Boolean predicate of the filtering expression.

REFERENCES

- [1] F. Risso, “NetPDL language specification,” February 2007. Available at <http://www.nbee.org/netpdl/>.
- [2] F. Risso, and M. Baldi, “NetPDL: an extensible XML-based language for packet header description,” In *Elsevier Computer Networks* Volume 50, Issue 5 (April 2006), pp. 688-706.
- [3] Computer Networks Group (NetGroup) at Politecnico di Torino, “The NetBee Library,” August 2004. Available at <http://www.nbee.org/>.
- [4] Computer Networks Group (NetGroup) at Politecnico di Torino, “Analyzer 3.0,” March 2003. Available at <http://analyzer.polito.it/>.
- [5] F. Risso, and M. Baldi, “A framework for rapid development and portable execution of packet-handling applications,” In *Proceedings of the 5th IEEE International Symposium on Signal Processing and Information Technology*, December 2005, pp. 233-238.
- [6] M. Baldi, F. Risso, “Towards Effective Portability of Packet Handling Applications Across Heterogeneous Hardware Platforms”, In *Proceedings of the 7th Annual International Working Conference on Active and Programmable Networks*, Sophia Antipolis, France, November 2005.
- [7] S. McCanne, and V. Jacobson, “The BSD packet filter: A new architecture for userlevel packet capture,” In *Proceedings of the Winter 1993 USENIX Conference*, January 1993, pp. 259-269.
- [8] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis, “FFPF: Fairly Fast Packet Filters,” In *Proceedings of 6th Symposium on Operating Systems Design and Implementation*. OSDI’2004, December 2004, pp. 347-363.
- [9] A. Begel, “Applying General Compiler Optimizations to a Packet Filter Generator”. 1996. Available online at <http://www.microolap.com/downloads/pssdk/literature/begel96applying.pdf>
- [10] R. J. Clark, M. H. Ammar, and K. L. Calvert. “Multi-protocol architectures as a paradigm for achieving inter-operability,” In *Proceedings of IEEE INFOCOM*, April 1993, pp. 136-143.
- [11] A. Begel, S. McCanne, and S. L. Graham, “BPF+: exploiting global data-flow optimization in a generalized packet filter architecture,” In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols For Computer Communication*, SIGCOMM ’99, September 1999, pp. 123-134.