POLITECNICO DI TORINO

SCUOLA DI DOTTORATO

Dottorato in Ingegneria Informatica e dei Sistemi – XXI ciclo

Tesi di Dottorato

Enabling Flexibility in High-Speed Packet Processing

Olivier Morandi

Tutore
prof. Mario Baldi

Coordinatore del corso di dottorato
prof. Pietro Laface

Marzo 2009

Author Publications

Parts of the research presented in this thesis have been previously published or presented in the following papers:

- Morandi, Olivier; Risso, Fulvio; Baldi, Mario; Baldini, Andrea, "*Enabling Flexible Packet Filtering Through Dynamic Code Generation*". In Proceedings of IEEE the International Conference on Communications, 2008. ICC '08. , pp.5849-5856, 19-23 May 2008
- Morandi, Olivier; Risso, Fulvio; Rolando, Pierluigi; Hagsand, Olof; Ekdahl, Peter, "*Mapping packet processing applications on a systolic array network processor*". In Proceedings of the International Conference on High Performance Switching and Routing, 2008. HSPR 2008. , pp.213-220, 15-17 May 2008
- Morandi, Olivier; Risso, Fulvio; Valenti, Silvio; Veglia, Paolo; "*Design and implementation of a framework for creating portable and efficient packet-processing applications*". In Proceedings of the 7th ACM international Conference on Embedded Software (Atlanta, GA, USA, October 19 - 24, 2008). EMSOFT '08.
- Morandi, Olivier; Risso, Fulvio; Moscardi, Giorgio; "*An Intrusion Detection Sensor for the NetVM Virtual Processor*". In Proceedings of the The International Conference on Information Networking 2009 (ICOIN 2009), Chiang Mai, Thailand, January 2009.

Table of Contents

1. Introduction

During last years we have assisted to an exponential growth of the Internet, both in the number of connected users and in the variety of services made available by an always increasing number of subjects. Indeed, the Internet is day-by-day more pervasive in our lives and we are gradually transferring to this “virtual world” many tasks and activities that until a few years ago were mostly peculiar to other domains.

On one side, we have seen the explosion of the World Wide Web and its shift from a one-to-many to a many-to-many paradigm, leading to the rise of successful phenomena like weblogs and social networks. This is coupled with another parading shift, in which the web is seen as a distributed platform providing what are called “web-services”. On the other side, we have seen the rise of several new possibilities for communicating, sharing and exchanging informations, and it is nowadays common to use Voice over IP (VoIP) services, P2P file-sharing, Internet radios and TVs, online applications (e.g. Google Docs). The network is increasingly used as a transport layer that allows distributing and exchanging complex and semantically-rich information.

In such scenario, Network Operators face several challenges, because they need to provide users with appropriate Qality of Service (QoS) and deploy adequate security

policies, while traffic loads on edge and core networks are increasing, and network procols are evolving in order to support newborn services. This pushes for the need of some degree of flexibility also in high-speed networking devices, like switches, routers and firewalls. So the Networking Industry must cope with extremely diverging requirements: on one hand, networking equipments must be able to keep up with line rates that are rising in the order of tens of gigabit per second, while, on the other hand, there is the need to shorten up the development cycle, in order to support novel protocols and advanced functionalities within shorter delays. In particular, this second point is being pushed to the limit of giving customers the ability to independently implement new functionalities, for example adding support to custom and proprietary protocols. As a direct consequence, the design of network devices can no longer rely on completely hardware-based solutions - usually employing Application Specific Integrated Circuits (ASICs) - for achieving high-throughput perfomances, because of the need of providing some degree of flexibility and programmability.

The need of accomodating both flexibility and performance requirements is common in many fields related to the design of embedded systems [1], and a widely adopted solution is to integrate several, possibly heterogeneous, processor cores on a single chip, along with specialized hardware coprocessors, in order to build what is called a System on a Chip (SoC). In particular, during the last ten years, chip vendors have been proposing several commercial Application Specific Instruction Processors (ASIPs) targeted to high-throughput packet processing, which are commonly known as Network Processors, or Network Processing Units (NPUs).

NPUs are specially designed and optimized for packet processing operations and, in order to achieve high througput performances, they usually provide from tens to hudreds of concurrent processing elements, which enable the exploitation of the intrinsic parallelism found in packet processing applications.

However, the expected advantages of using Network Processors for designing today's network devices, come at some costs in terms of ease of development of software applications. In particular, difficulties are mainly tied to the high heterogeneity of available architectures, making nearly impossible the development of portable applications. In fact, a program written and optimized for a specific NPU cannot be retargeted to work on a different hardware platform, because, unlike for general purpose processors that expose a coherent programming model, network processors generally expose a variety of heterogeneous low-level programming models, spanning variably from shared-memory multi-processing models to pipelining and message passing ones. Moreover, there is the lack of a standard high level programming language for these processors, as often vendors provide a software development kit, which uses a C-like language with extensions that are peculiar to the specific hardware platform. It is also common to write applications directly in the native assembly language of a given network processor.

The depicted scenario highlights the need for novel solutions capable of increasing the reuse of software components and to shorten the development cycle in the design of complex packet processing applications, while still ensuring the fulfillment of perfomance constraints. The aim of this work is to respond to such defy, by taking into account two main aspects of the problem: (*i*) the need of enhancing the portability of software solutions, achieving a looser coupling between packet processing programs and the specific hardware platforms where they will be executed, and (*ii*) the need of being able to adequately follow the evolution of network protocols, by allowing applications to seamlessly incorporate the support to emerging ones without the additional costs related to stepping into a new development cycle (i.e. program refactoring, quality assurance, release).

The former point is addressed by investigating the opportunity of capturing the intrinsic characteristics of packet processing applications into a novel programming model, which is able to adequately abstract the functionalities usually provided by network processor architectures, while allowing platform-specific mapping choices to be isolated in a set of back-end modules. The latter point is addressed by investigating the possibility of decoupling the logic of applications from the knowledge of network protocols, providing the user with a set of languages and tools that allow writing protocol-agnostic packet processing software with performances that are comparable to those of hand-written programs.

This work is structured in two main parts. In the first one, two ortogonal solutions for enabling flexibility in packet processing software are analyzed: in Chapter 2 the NetVM programming model is presented as a possible solution for creating both portable and efficient packet processing applications, while Chapter 3 will introduce the languages and the outline of an architecture for obtaining efficient protocol-agnostic applications. The second part aims at validating the proposed solutions. In particular, Chapter 4 will present the implementation of the NetVM model as a runtime environment with a multi-target compiler infrastructure; in Chapter 5 the capability of NetVM to support the development of complex packet processing applications is assessed; Chapter 6 presents the architecture of a compiler for the dynamic generation of packet filtering programs based on an external protocol description database; experimental results are reported in Chapter 7, while in Chapter 8 conclusions are drawn and future work is outlined.

PART I. Enabling Flexible Packet Processing

2. Towards Portable and Efficient Packet Processing Applications

2.1. Introduction

During the last decade, the increasing requirements in terms of flexibility for the design of high-speed networking devices have pushed the Industry towards the development of network processors, i.e. programmable processors providing several concurrent execution units, with Instruction Set Architectures (ISAs) specifically targeted to packet processing, usually integrated with special purpose hardware coprocessors for offloading computational intensive functionalities. Even though such devices are not able to achieve the same throughput performances of ASIC based chips, they provide more flexibility thanks to their programmability.

However, network processors have traditionally exposed several problems from the point of view of the ease of programming. Such problems mainly relate to the need for the programmer to deal with very low-level aspects of the hardware, and to the difficulty of manually partitioning application modules across several concurrent

processing engines. The tools and Software Development Kits (SDKs) provided by manufacturers are in most cases a partial solution, since they tend to expose the hardware to the programmer through an assembler language, and even when a high level language such as C is provided, it is generally extended with constructs that directly maps onto hardware features, leading to a lack of abstraction.

Beside problems related to programmability, network processors suffer from a major problem given by the impossibility to reuse software solutions across different hardware platforms. Applications that have been developed and optimized for a specific NPU architecture, when needing to be ported to a different architecture, must be redesigned from scratch and have to follow again the entire development cycle. Indeed, network processor architectures proposed from different vendors are extremely heterogeneous between them. They span from symmetric multi-processing platforms like the Intel IXA family (IXP12XX, IXP24XX, IXP28XX) [2] and the more recent Cavium Octeon [3] network processors, to systolic array dataflow processors like the Xelerated X11 [4] and the Bay Microsystems Chesapeake [5], which are capable of processing packets at speeds in the order of tens of gigabits per second. A survey on the characteristics of some of these NPU architectures can be found in Appendix 0.

Given such high heterogeneity, the problem of defining a common programming model, capable of providing *generality* (i.e., capability to support a wide range of applications), *portability* (across a wide range of network processor architectures), *efficiency*, while still providing an *adequate programming abstraction*, is particularly difficult. As will be detailed in Section 2.2, current solutions usually provide the latter two features, but no solutions exist that looks at the problem in a comprehensive manner. In particular, the generality of the approach and portability are usually not taken into account, since the proposed solutions are mainly targeted to specific hardware architectures, or are able to accommodate a very specific class of applications.

Portability and efficiency are usually considered as conflicting requirements that can be hardly achieved altogether in a specific solution. Indeed, while the introduction of an abstraction layer capable of hiding the differences between different hardware platforms can represent the basis for enabling the creation of portable software, the achievement of adequate performances from the same application executed on a wide variety of heterogeneous architectures is extremely challenging.

The main argument of this thesis is that, in the case of network processing software, portability and runtime efficiency can be achieved both at the same time. In fact, packet-processing applications are usually very limited in scope and expose very recognizable structural and behavioural patterns. Such characteristics that are peculiar to the specific application domain can be exposed to the programmer through an adequate programming model, and inside the intermediate representation of a multi-target optimizing compiler, allowing an efficient mapping on heterogeneous architectures and enabling the deployment of aggressive special purpose optimizations.

In order to support this argument, part of this thesis work has been devoted to refining and validating the concept of a Network Virtual Machine (NetVM) [6][7], previously proposed by the NetGroup from Politecnico di Torino. NetVM provides a mid-level abstraction layer, based on a dataflow programming model in which hardware is virtualized, with the result of completely hiding the target architecture to the programmer. In other words, NetVM aims at applying the well-known paradigm “Write once, run everywhere” proposed by the Sun Java Virtual Machine (JVM) [8] and the Microsoft Common Language Runtime (CLR) [9] to the field of network processing software, where performance is a key factor.

One of the main objections to this approach is that the introduction a common abstraction layer, while enabling portability, would result in a substantial overhead, wasting the benefits of using special purpose and optimized hardware architectures. The

first part of this thesis will demonstrate that this claim is not necessarily true in the case of a virtual machine specifically designed for packet-processing applications, like the NetVM. In particular, the NetVM model exposes a set of key features that, besides making it a good target for different high-level languages, enable both portability and an efficient mapping on the target hardware.

After a brief overview of the available related work, the current chapter will present the NetVM abstraction layer, analyzing the points that make it a good choice for developing portable and efficient network data-plane applications, while the following chapters will focus on the implementation of the model as a portable runtime environment and multi-target optimizing compiler. In chapter 0, experimental results will show that NetVM applications can be efficiently executed, without any change, on three different platforms such as the Intel x86 general purpose architecture, the Cavium Octeon [3] multi-core network processor and the Xelerated X11 [4] systolic array processor.

2.2. Related Work

In the last years, the problem of creating a suitable framework for programming network processors has been widely investigated from both industry and academia.

Click [10] is a framework for implementing a modular software router, by interconnecting different packet processing modules called elements. Elements implement specific functions like packet classification, queuing, scheduling, and interfacing with network devices. A router configuration is built connecting elements in a directed graph, which represents the flow of the packets inside the router. A Click element is written in C++, and is a subclass of the virtual class Element. NP-Click [11] proposes a programming model based on the Click language for Intel IXP network processors, showing that the level

of abstraction introduced, while easing application development, also enables an efficient mapping on a special purpose architecture.

Memik et al. [12] demonstrate the advantages of structuring applications for network processors in a modular way, and describe a system, called NEPAL, which is able to extract the modules that constitute a sequential network-processing program for mapping them on parallel execution units.

PPL (Packet Processing Language) [13], defined by IP Fabrics Inc., is a declarative language for programming network processors of the Intel IXA family. A virtual machine executes PPL programs on the target platform, and is in charge of mapping high level constructs onto the available hardware features, enabling the transparent exploitation of parallel processing engines. While the details of this solution are unknown, the fact of being highly tied to a specific high-level language (i.e. PPL), and to a specific platform (Intel network processors), represents a limitation.

Wagner et al. in [14] proposed a C compiler for an industrial network processor, showing that exposing low-level details in the language through compiler known functions allows an efficient exploitation of the available hardware features without relying on assembly language.

PacLang, by Ennals et al. [15] is a framework that allows application designers to use a simple high-level language to partition packet processing programs in different concurrent tasks. The base elements of PacLang are tasks and queues. Tasks represent computations that can be executed concurrently, while queues can be assimilated to pipes, through which tasks can communicate and synchronize. The proposed system allows a PacLang program to be automatically partitioned on parallel execution units; however, its capabilities have been demonstrated only on Intel IXP network processors.

Shangri-la [16] is a system with a more general approach. Its basic components are a domain-specific programming language (Baker) and a profile-guided compiler

infrastructure, which is able to optimize and map an application onto Intel network processors. While the reported performance results look promising, it is unclear if the compiler framework can be retargeted in order to support different architectures. Moreover, the solution appears to be tied to the Baker language.

These approaches generally fail to provide a comprehensive framework for achieving at the same time both efficiency and portability across heterogeneous architectures. In particular, those that are targeted to a specific platform, focusing on performance, tend to expose in high-level programming languages a set of low-level primitives very close to the hardware, causing a lack of abstraction. For example, programming models targeted to multi-core based network processors may include explicit primitives for task/thread synchronization, while other may provide special functions for accessing coprocessors or that are directly mapped onto special-purpose instructions. These can be present either in the form of library functions and APIs, or intrinsics (i.e. compiler known functions). In such scenario, the model is hardly portable because it is too much tied to the target architecture and porting it to another platform may be too much costly if not impossible, like for example when needing to map thread synchronization on a systolic array processor. Additionally, this approach, which originates from a bottom-up vision, tends to prevent programmers from having an abstract vision of their application, because they are forced to structure the software according to the execution model supported by the hardware, e.g. by defining the appropriate task/thread partition and dealing with synchronization issues explicitly, hence preventing any possible outcome in terms of portability.

Vice versa, the approaches that are more application-oriented usually tend to completely hide the details of the underlying hardware, possibly enabling software portability. However, they usually lack in generality since they are mostly tied to a specific class of applications, leading to the impossibility to effectively use the model for writing different kinds of applications, with the result of actually limiting the flexibility of the approach.

In contrast to previous solutions, the one proposed in this thesis is based on a virtual machine specially targeted to packet processing applications and aims at providing a comprehensive programming model that is able to deliver high performances on target architectures supporting it, while ensuring complete code portability and generality (i.e. the capability to support several kinds of applications) through a mid-level abstraction layer. This result is achieved by completely hiding the details of the hardware to the programmer and by capturing in the programming model the characteristics that are peculiar to the network-processing domain, allowing the compiler to have a more detailed view on the semantic of the application, thus enabling an efficient mapping.

2.3. Using a Virtual Machine for Code Portability

The concept of a virtual or abstract machine is commonly used when facing the need of hiding from the programmer the characteristics of the real execution units where programs will be actually executed. This decoupling allows the same programs to be executed on any system where an implementation of the abstract machine is available. In particular, the implementation of a virtual machine usually comprises a component called the Runtime Environment, which provides a mapping of the abstract components onto the target architecture, and a component for translating into executable code the instructions of the source program. The latter can be an interpreter, an ahead of time (AOT) compiler, or a just-in-time (JIT) compiler. Figure 1 shows these three possible scenarios.

In the case depicted in Figure 1A, the source program is executed through an interpreter, i.e. a program that is able to decode a sequence of instructions and execute them by emulating their behaviour. Because of its simplicity, usually the implementation of scripting languages and the reference implementation of most virtual

machines falls in this scenario. However, such this approach is not able to provide adequate runtime performances, since the process of decoding source instructions and emulating them is very time-consuming.

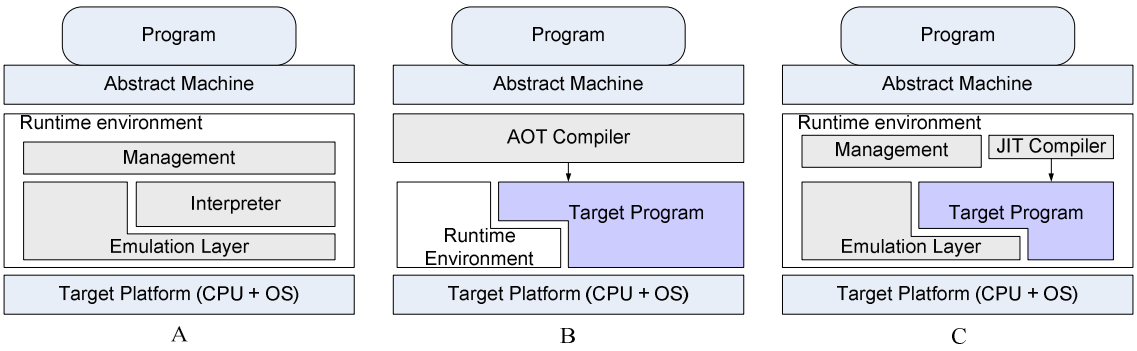


Figure 1. Three different implementation schemes for abstract machines

Figure 1B shows the scheme used when runtime performances play a major role. In such case, the source program is translated ahead of time into a program that can be directly executed on the target machine, possibly exploiting functionalities provided by a runtime environment. The compilation process happens once, while the execution of the target program is delayed and can be repeated several times, e.g. with different inputs. This implies that the complexity of the code generation and optimization techniques featured by the compiler can be tuned, based on the required runtime performances. This scenario is common when implementing traditional high-level programming languages like C/C++, etc. Even though it is quite uncommon to associate a language like C to virtual machines, the reader should note that every computer language at any level of abstraction also defines an underlying abstract machine that is able to execute its primitives [17].

The scheme shown in Figure 1C is the one typically employed for the implementation of modern programming language virtual machines such as the JVM and the CLR. Here, the source program is translated into native code for the target platform just before execution, through a just in time compiler. Indeed, in order to limit

the delay due to the compilation time, usually the source program is not compiled all at once, but only one module at a time, when needed, during execution. This schema sits between the former two, allowing the direct execution of source code on the target machine, with better runtime performances respect to the use of an interpreter. However, especially for general-purpose applications where the user is directly involved, a program compiled just in time will necessarily provide poorer performances respect to an equivalent program compiled with a full-featured AOT compiler. In order to guarantee an appropriate user experience, a JIT compiler must perform a compromise between the required compilation time and the quality of the generated code, which are two parameters tied by an inversely proportional relation: for obtaining better target code (e.g. providing more processing speed), the compiler should perform more complicated and aggressive optimizations, that in turn would increase the compilation time.

For this last reason, it is common to associate the concept of a virtual machine with poor runtime performances, because general purpose interactive programs, either being interpreted, or compiled just in time, are generally slower than equivalent programs compiled with a full optimizing ahead of time compiler.

If such consideration can be true for general-purpose virtual machines like the JVM and the CLR, it does not necessarily apply to the case of a domain specific virtual machine especially designed for the development and execution of packet processing applications. In particular, we should note that networking data-plane applications expose an execution pattern very different from that of typical general-purpose applications, minimizing, in the former case, the differences between ahead of time and just in time compilation. Moreover, as will be detailed in the rest of the chapter, the definition of an appropriate model for the abstract machine can enable the deployment

of special purpose optimizations, not applicable in compilers for general-purpose languages.

In other words, the term “Virtual Machine” does not directly imply anything about runtime performances, since these are mainly tied to the peculiar characteristics of the abstraction it provides. In fact, the supposed slowness of Java programs on general-purpose hardware is partially due to the features provided by the JVM abstract machine. In particular, Java programs are always executed in a safe sandbox, guaranteeing that no out of bound memory accesses will compromise the host machine. Moreover, Java uses implicitly a garbage collector for deleting unused objects, freeing the programmer to deal with manual memory allocation/deallocation. These and other features come at an additional cost in terms of runtime performances, especially when using just in time compilation, where the deployment of sophisticated optimizations is discouraged.

2.4. JIT Compilation of Networking Data-Plane Applications

As already said, in general purpose virtual machines like the JVM and the CLR, the use of just in time compilation responds to the need of enhancing the performances perceived by the user of interactive applications, however Pletzbert and Cytron [18] point out that in such context, the JIT compilation of Java applications does not always guarantees better performances than those obtained through a bytecode interpreter:

“While the just-in-time approach avoids the penalty of interpretation, our experiments show that the cost of compilation can significantly interrupt the flow of execution; furthermore, in many cases, better performance could be obtained by interpreting the original form rather than compiling to native code.”

In particular, since JIT compilation of a program module is usually performed right before its execution, the main constraint that must be satisfied for JIT compiled code to be more efficient than interpreted code is the following:

$$T_{Int} > T_{Jit} + T_{Exec} \qquad (1)$$

where T_{Int} is the time taken for interpreting a program module, T_{Jit} the time taken by the compilation process, and T_{Exec} the time spent during the execution of the resulting machine code. Indeed, the translation to native code always introduces a delay in the execution of a program module and, in order to maximize performances, both T_{Jit} and T_{Exec} have to be minimized. On the other hand, such parameters are not unrelated, since the quality of the machine code generated, and consequently its speed, highly depend from the quality of the compiler and from the possibility to apply aggressive (i.e. more costly) optimizations. A more complex JIT compiler can produce machine code that can run faster than the code generated by a simpler one, but if the increased complexity can reduce T_{Exec}, at the same time makes T_{Jit} bigger.

In other words, engineering a just in time compiler for a general purpose virtual machine means searching a satisfactory compromise between compilation time and the quality of the generated code. In the last years, several solutions for both the JVM and the CLR have been proposed. Many of them try to reduce T_{Jit} sacrificing the opportunity of applying aggressive optimizations [19][20][21][22]. The code generated by these compilers is of average quality and is usually well suited for general-purpose applications, where time constraints are not critical. Besides, in order to reduce the latency of execution due to the compilation process, solutions like “continuous compilation” have been proposed [18]: while a program is being compiled by the JIT, the interpreter begins executing it, until the control of execution can be transferred to the generated machine code.

However, such considerations hardly could be applied in the domain of packet processing applications, whose only purpose is to process (possibly infinite) sequences of network packets, and which are by nature not interactive at all. In particular, for a packet-processing module, the constraint imposed by (1) takes the following form:

$$nT_{Int} > T_{Jit} + nT_{Exec} \qquad (2)$$

where n is the whole number of packets processed during the life of the application.
The same equation can be rewritten as:

$$T_{Jit} < n(T_{Int} - T_{Exec}) \qquad (3)$$

The result is that if the JIT compiled code performs better than the interpreted one and for n large enough, the constraint is always satisfied. This means that, due to the non-interactive nature of packet processing applications, as for any other kind of data-intensive software, the pure cost of just-in-time compilation, for large it would be, is a factor that does not directly influence the perceived performances, and the only constraint is that the native code generated by the compiler should be faster than the interpreted one.

However, the main outcome of such considerations goes over the simple comparison between the performances of JIT compiled versus interpreted code. Indeed, it is clear that the major term to be considered for the design of a Just in Time compiler for a packet processing virtual machine is the quality (i.e. usually measured by the speed) of the generated code. In other words, it is not necessary to trade compilation time for runtime performances, leading to a situation very close to the one of ahead of time compilation, which, being performed completely offline allows the deployment of extremely aggressive optimizations.

2.5. The Network Virtual Machine

The NetVM abstraction layer [6][7] defines a dataflow programming model for data-plane networking software, where an application is expressed as the interconnection of a set of independent packet processing modules called Network Processing Elements (NetPEs). Indeed, a NetVM application can be viewed as a directed acyclic graph, whose nodes represent NetPEs, and whose edges represent connections between consecutive modules. NetPEs are interconnected between them through *Ports*. Network packets are like tokens that flow through the graph from a source to a sink, while being processed by NetPEs. In particular, packet sources and sinks are called respectively input and output *Sockets*, which can be connected to both physical network interfaces and "application interfaces" that allow packets to be injected by, or sent to user-defined control-plane modules.

The use of a dataflow model for expressing networking applications is not novel and is quite common [10][12][16]. This stems from the consideration that such kind of applications can be described as a collection of relatively independent tasks to be performed on packets; once a module has finished processing a packet, this can advance toward the next one, and the first is ready to accept a new packet, following a pipelining schema.

Actually, the entity flowing through NetPEs is not a simple network packet, but a more complex structure called *Exchange Buffer*, which, besides the packet buffer, contains additional information, like a timestamp and a special memory buffer called the *Info Memory*, which consecutive NetPEs can use for exchanging data associated to the packet.

Figure 2 shows an example of a generic NetVM application, viewed as a collection of interconnected NetPEs.

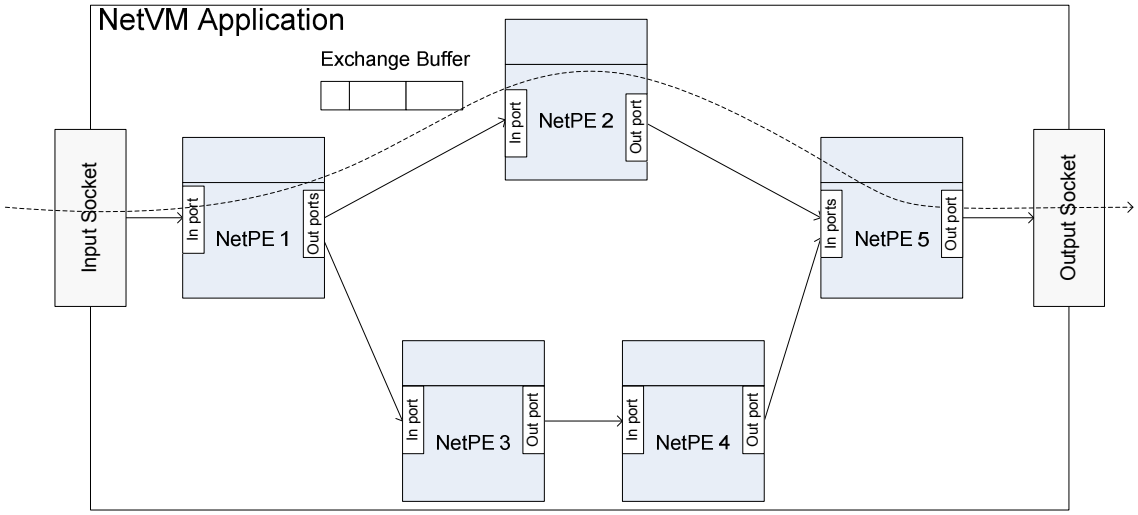


Figure 2. NetVM application viewed as a dataflow graph of NetPE modules

Each NetPE exposes a user-defined number of ports, through which exchange buffers can be either received or sent. The ports of each NetPE can be connected to ports of other NetPEs or to Sockets. Both input and output ports are classified in two categories: push and pull. Based on the class of the two ports involved in a connection, the way in which exchange buffers are transferred between consecutive NetPEs varies. In particular, in a push connection the packet is “pushed”, i.e. sent from an upstream to a downstream NetPE, while in a pull connection the packet is “pulled”, i.e. requested by a downstream NetPE to an upstream one.

Due to its dataflow nature, NetVM follows an event-driven paradigm, so the behaviour of a NetPE module is defined by specifying a set of event handlers that are executed when specific events happen. In particular, the NetVM model defines three main events that each NetPE should handle, that are (*i*) NetPE initialization, (*ii*) the arrival of an exchange buffer on an input push port, and (*iii*) the arrival a request to send an exchange buffer on an output pull port. The corresponding event handlers are named respectively Init, Push and Pull. In particular, the Init handler of each NetPE is triggered once, before starting packet processing, and allows the private state of the NetPE to be

initialized, while the Push and Pull event handlers express the actual tasks to be performed in order to process packets.

NetPE event handlers can be programmed in a mid-level language called Network Intermediate Language (NetIL), which is a stack-based assembler providing an instruction set specifically targeted to packet-processing applications.

The choice of making NetIL a stack-based language, in contrast to traditional register-based schemas, is simply dictated by the fact that the implicit presence of an operand stack avoids the necessity to assign explicit names to the temporary results of operations, leading to a more compact binary representation. Indeed, the actual expressivity of a stack-based language is equivalent to that of a register-based one.

On the other side, the choice of defining a mid-level assembly language stems from the need of making NetVM general enough to be independent from any specific high-level language. In fact, NetIL can be an excellent target for several high-level languages, ranging from declarative (e.g. rule based), to imperative ones (e.g. like C). More details on the characteristics of NetIL will be given in section 2.5.4

2.5.1. NetIL Execution Model

As for any computer language, NetIL defines its own execution model, where the abstract architecture of the NetPE, shown in Figure 3, plays a major role.

A NetPE is a 32-bit stack-based processor that is able to perform integer operations on data stored in a set of local memories. Floating-point operations are not supported, because they are generally not used in packet processing applications. A local processing unit executes the instructions stored into the code memory, which contains the three NetPE event handlers (i.e. Init, Push, Pull). The starting addresses of the handler programs inside the code memory are available in three read-only registers, named respectively INA (init handler address), PSA (push handler address), and PLA

(pull handler address), so, once a particular event occurs, the correct program is executed.

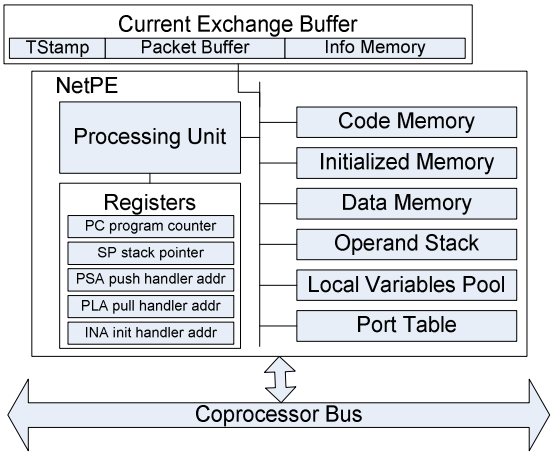


Figure 3. NetPE Internal Architecture

Instructions operate on values loaded onto the operand stack, and results are pushed onto the stack as well. A set of local variables allows storing temporary data that is guaranteed to survive only until the end of the current handler being executed.

Every NetPE can access one or more “virtual coprocessors”, for executing complex operations, such as lookup and regular expression matching, which are likely to be implemented in hardware on network processing platforms. More details on NetVM coprocessors will be given in Section 2.5.5.

2.5.2. Memory Layout

NetVM provides a rich memory model, whose structure stems from the following considerations on typical packet processing applications:

The packet is the fundamental entity that is central to the whole application and needs to be explicitly identified

Although different NetPEs represent relatively independent tasks to be performed on packets, it is frequent that a module needs to communicate to subsequent ones partial results, in the form of single values or structured data

Persistent or static state (e.g. forwarding tables, lookup tables, counters, etc.), is usually localized into a single module, and, provided a communication system based on the previous points, there is no need of shared state among different modules

The result is a set of orthogonal memory segments that reflect the needs of the programmer for storing temporary or persistent state, and for communicating values between different modules of a packet-processing application. In particular two memory segments flow through modules carried inside exchange-buffers, i.e. the packet buffer and the info memory, while a memory that is local to each module, i.e. the data memory, allows storing static data that should survive across consecutive executions of NetPE event-handlers.

The size of the info and data memory segments can be defined through specific directives in the source NetIL assembly, while the size of the packet buffer is initialized to the actual length of the incoming network frame, when an exchange buffer is created and injected into the application. The virtual machine ensures that no memory access is performed out of each segment boundaries.

NetVM does not provide any explicit mechanism for memory allocation and deallocation. All memory segments are statically allocated, either in the initialization phase, or at the creation of an exchange buffer. This choice is mainly dictated by performance constraints, since memory allocation and deallocation at runtime may be costly on some architectures, and by the consideration of the fact that in real-world packet processing applications, persistent and complex data structures (e.g. a forwarding table) are usually created by the control plane (e.g. through a routing protocol process or manual configuration) and consumed in a read-only fashion by the data-plane program. On the other hand, if a NetVM application would need a memory allocation mechanism for managing complex memory structures that must be updated at runtime, such as for example a session table, the programmer should consider to abstract the complex

functionality (e.g. a lookup engine) through a virtual coprocessor (see Section 2.5.5), enabling an efficient mapping on a wider variety of target platforms.

2.5.3. Threading model

Even if NetPEs can be viewed as a set of concurrent packet processing tasks, NetVM is based on a purely sequential threading model. In particular, the execution of a NetPE packet handler is tied to the presence of an exchange buffer, and only one exchange buffer is allowed to be processed by a NetPE at a specific time; on the other hand, an exchange buffer cannot be associated to more than a NetPE at a time. This stems from the dataflow nature of the NetVM model, for which each exchange buffer traverses a pipeline of NetPEs during its journey through the application, triggering the execution of a sequence of packet handlers. In other words, for a given exchange buffer, a specific instruction path of the application is executed sequentially.

Complementarily, each NetPE during its operation “sees” a sequence of exchange buffers, and ideally, at a given time, every NetPE should be processing a different exchange buffer.

2.5.4. NetIL Instruction Set

The NetVM instruction set derives from the one of a generic stack machine with additional instructions to support packet processing. Instruction opcodes can be subdivided into several groups; the most important ones are listed in Table 1.

In NetIL the only supported data type is 32 bit unsigned integer, although signed variants of arithmetic operators are available, ensuring a correct handling of overflow and underflow conditions. Memory accesses can be performed on 8, 16, and 32 bit locations, and each value loaded from memory is either zero or sign extended to 32 bit, depending on the type (signed or unsigned) of the memory read instruction. On the other hand, since the operand stack is 32 bit wide, 8 and 16 bit memory stores are

performed by truncating a 32 bit value on a byte or word boundary, keeping the least significant bits.

The highly structured layout of NetVM memories is reflected in NetIL, where, for every kind of memory (packet, info, data), there is a specific group of access operators. Since numeric data in network packets is stored in network byte order (i.e. big endian), packet memory read and writes of 16 and 32 bit values perform an implicit network-to-host byte order conversion; on the other hand, data is stored in the info and data memories in host byte order, i.e. the natural byte order of the target machine. While byte ordering does not affect the internal functioning of the virtual machine (since conversion is automatically performed when loading and storing data), this is important when looking at the interaction of the NetVM with the outside world. In other words, an external program using the NetVM must provide it a packet buffer formatted in network-byte order, while a simple read from the internal memory of the NetPE (if needed) will expect to find data in the host byte order.

The NetIL instruction set provides operators that are frequently used in packet processing applications and that are likely to be implemented in hardware in network processing architectures, like for example bit manipulation instructions. Besides, a wide variety of flow control operators is available; since the main purpose of packet processing programs is to take decisions based on the content of network packets, the usual jump and branch instructions are provided, as well as the more complex field comparison operators and a multi-way branch (i.e. the switch/case construct). The latter is particularly effective for implementing protocol demultiplexing (i.e. deciding which is the next protocol header based on the value of a specific field).

Table 1. NetIL instruction set summary

Category	Examples	Description
Arithmetic and Logic	`add, sub, mul, neg`	Basic arithmetics
	`shl, shr, rol, ror`	Shift and rotate
	`and, or, xor, not`	Bitwise logic
Bit Manipulation	`set.bit, clear.bit, flip.bit, test.bit`	Bit test, set, flip and clear
	`clz`	Count leading zeros
	`find.bit`	Find the first bit set
Flow Control	`jump, jcmp.eq, jcmp.neq, jcmp.l, …`	Jump and branches
	`switch`	Switch/Case construct
	`call, ret`	Procedure call and return
Locals	`locload, locstore`	Local variable load and store
Memory Access	`pload.8, pload.16, pload.32`	Packet memory load
	`pstore.8, pstore.16, pstore.32`	Packet memory store
	`iload.8, iload.16, iload.32`	Info memory load
	`istore.8, istore.16, istore.32`	Info memory store
	`mload.8, mload.16, mload.32`	Data memory load
	`mstore.8, mstore.16, mstore.32`	Data memory store
Field comparison	`jfield.eq, jfield.ne, jfield.lt, …`	Packet buffer comparisons
Packet transfer	`pkt.send, pkt.receive`	Packet send and receive
Stack management	`push`	Push constant
	`pop`	Discard top of the stack
	`dup`	Duplicate top of the stack
Coprocessor Interaction	`copro.in, copro.out`	Coprocessor reg read/write
	`copro.invoke`	Invoke coprocessor operation
	`copro.init`	Coprocessor initialization

2.5.5. Coprocessor Abstraction

Since packet-processing applications usually rely on a set of functionalities that are often implemented directly in hardware on many network processor architectures (e.g., Content Addressable Memories for fast table lookups, hashing, string matching, etc.), the NetVM model includes the concept of virtual coprocessors, i.e. a way to make such features available to the programmer through a well-defined interface. A coprocessor is viewed by the application as a black box providing specific operations; while its coherent interface guarantees the portability of the software on different platforms, its implementation may vary from platform to platform. In particular, on architectures that do not provide any hardware acceleration, coprocessors should be emulated by software, while on architectures providing special purpose features, coprocessors may be mapped directly on hardware.

From a logical point of view, a NetVM coprocessor is composed of a set of directly addressable 32 bit registers and a local processing unit, as shown in Figure 4. NetVM Coprocessor Architecture. Registers can be accessed through the NetIL instructions `copro.in` and `copro.out`, while the instruction `copro.invoke` is used for triggering the execution of a specific operation from the processing unit.

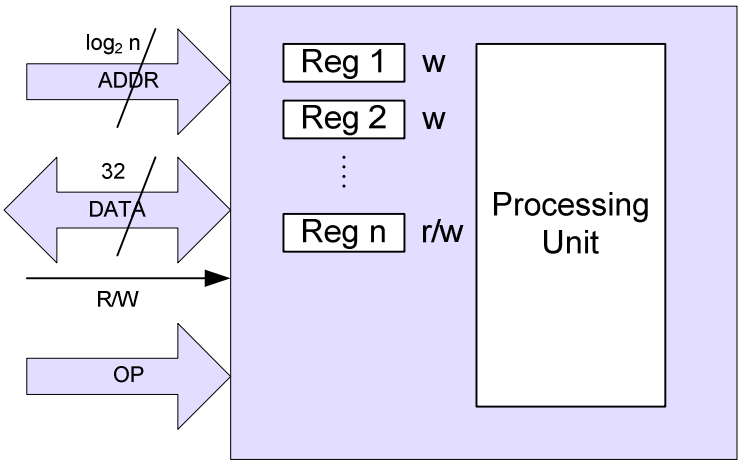


Figure 4. NetVM Coprocessor Architecture

In particular, the operations to be performed for executing a coprocessor function are the following:

Write the appropriate values into the coprocessor input registers through the `copro.out` instruction

Invoke a coprocessor operation through the `copro.invoke` instruction

Read the result from one or more coprocessor output registers through the `copro.in` instruction

Since some complex functionalities, such as the ones for regular expression matching, need to access the entire packet buffer (e.g. for scanning the payload in search of a string), the NetPE can release the exchange-buffer and pass it to a specific coprocessor through the `copro.exbuf` instruction.

Coprocessors may also support an initialization phase that is invoked through the `copro.init` instruction.

2.6. Why NetVM Enables both Portability and Efficiency

The abstraction layer introduced by the NetVM exposes a set of key features that, while enabling the portability, also allow an efficient mapping of packet processing applications on extremely heterogeneous architectures. This is possible because the NetVM programming model favours the sharing of relevant information on the semantic of the application between the programmer and the compiler dedicated to mapping it on the target architecture. In particular, if the use of constructs borrowed from the application domain partially constrain the freedom of the programmer respect to the use of more general programming languages, like C, on the other hand, it allows the compiler to have a more detailed view on the intentions of the programmer, allowing it to perform a more efficient mapping, and to deploy more aggressive optimizations that a compiler for a general purpose language could not.

This Section analyzes how such concepts are captured in NetVM, and it will point out how the features of its programming model enable either the portability of user applications, either an efficient mapping on a wide range of heterogeneous hardware architectures.

2.6.1. Dataflow programming model

As described in Section 2.5, NetVM is based on a dataflow model of network data-plane applications, which can be usually described as a collection of relatively independent tasks to be performed on packets. This allows to make explicit the coarse-grained parallelism between functional modules, enhancing the possibility of efficiently

mapping the application on multi-core processors [16]. Besides, since the programmer has to deal with simple event handlers that are sequentially triggered by network packets flowing through application modules, it is quite easy understanding the logic of the software that can be viewed as the composition of self-contained functional blocks.

With respect to these points, other major programming models like the purely sequential and the parallel ones are both subject to different kinds of problems. The former one, even when modular and while being more natural for the programmer, exposes no relevant information to the compiler for extracting the coarse-grained parallelism between modules, so complex and possibly inefficient analyses need to be put in place in the compiler when needing to parallelize the code on multiple cores [23]. On the other side, classical parallel programming (e.g. multithreading), based on concurrent modules sharing state between them, poses different problems to both the programmer and the compiler, because (*i*) the task of protecting shared state against hazards is left to the developer through synchronization primitives like locks, semaphores and mutexes, leading to software that is hard to understand and maintain, and (*ii*) the applications based on concurrent tasks or threads, while can be easily mapped on multi-core environments providing hardware support for synchronization and locking primitives, may lead to an inefficient mapping on single core processors, and cannot be targeted to massively pipelined architectures, like systolic array network processors. Moreover, while the compiler can easily perform intra-module optimizations, the application of inter-module and application-wide global optimizations in concurrent programs can be extremely challenging.

The event-based nature of the dataflow paradigm, which mixes both communication and synchronization, allows to prune away many sources of non-determinism that are intrinsic in multi-threading, as noted in [24], with major advantages for both the programmer and the compiler. With respect to the latter, when the dataflow graph of

modules is acyclic (and in our case it is), it is extremely simple to translate it in a single control flow graph, obtaining a completely sequential program, by inlining consecutive modules.

From the point of view of portability and efficiency, this enhances the chances of mapping NetVM applications on extremely heterogeneous platforms, such as single core, multi core, or even systolic array network processors, without performance penalties, because the compiler has a complete view on the application, and can perform aggressive inter-module optimizations and, on multi-core architectures, apply any suitable strategy for automatic parallelization of sequential code, as those presented in [23].

2.6.2. Domain-Specific Intermediate Language

NetIL, the language employed for programming NetVM applications, has been designed to be general enough for being the ideal target for multiple high-level languages and, at the same time, for providing an adequate level of abstraction in order to allow the portability of packet processing software and an efficient mapping across several heterogeneous network processing platforms. In fact, NetIL has been profitably employed for the development of two high-level frontends, mainly for packet filtering and classification languages, which have been presented in [25][26].

In contrast, other frameworks for the development of efficient packet processing applications [13][14][15][16], tie their solution to a high-level programming language (e.g. domain specific, or derived from the C language), with the result of limiting the generality and the flexibility of the proposed approach.

As pointed out in Section 2.2, besides the mentioned problems from the point of view of generality, some of the solutions proposed by both the industry and the academia suffer also from the point of view of portability, because of the choice of incorporating in the high level language the features that are specific to the target

architecture, or to its low level programming model. For example, the IXA SDK, provided by Intel for programming the network processors of the IXP family, relies on a modified version of the C language, where key assembly instructions of the target ISA (Instruction Set Architecture) are exposed to both the programmer and the compiler as intrinsic functions (i.e. functions whose semantic is known by the compiler), with the result of tightly coupling the software to the specific architecture and preventing its portability. Another approach that may pose some problems for portability over heterogeneous architectures is the one proposed with PacLang [15], where the high level language exposes constructs representing tasks, queues and explicit synchronization primitives that are tied to the multi-threaded programming model of the target platform (i.e. the Intel IXP2400).

NetIL, instead of abstracting the hardware functionalities of a specific architecture, provides constructs that abstract the functionalities that are commonly needed by packet processing applications, making them available to the programmer. A backend compiler can then map them efficiently on the hardware features that the target platform may provide, with the result of enabling flexibility, portability and efficiency, all at the same time. At some extent, this can be viewed as a generalization of the approach proposed by [14], which presents a compiler for a modified version of the C language, where the packet manipulation functionalities commonly used by networking applications (e.g. packet access, bit manipulation, etc.) are exposed as intrinsics, which can be efficiently mapped on the target platform through the generation of the appropriate assembly instruction sequences, instead of relying on potentially inefficient library function calls. The key point that the two solutions share in common is the aim of rendering explicit in the source language the most common packet manipulation functionalities, as well as other features borrowed from the specific application domain, thus providing an adequate abstraction layer to the programmer and allowing the compiler to perform

more aggressive optimizations based on the knowledge of the semantic of such operations. Moreover, since NetIL also aims at being general enough to support different kinds of applications it is not tied to any specific high-level language and it adequately mixes low and mid level constructs, in order to be an effective target for several (possibly novel) high-level languages.

As a simple example, NetIL provides the *switch-case* construct, which is common in many high-level languages. In particular, the presence of such construct is very important, because it is widely used in packet processing programs for demultiplexing protocol headers, and making it explicit in the language allows a backend compiler to chose how implement it in the most efficient way on the target platform, for example by exploiting a TCAM-based lookup coprocessor, as Section 4.5.3.3 will show.

Another example is given by the field-comparison and bit-manipulation instructions of NetIL, which correspond to functionalities commonly used in packet processing programs. Even though it is likely that some NPU architectures provide similar instructions, the NetIL abstraction completely hides low level details from the programmer, who simply use those them as packet processing primitives, delegating to the compiler the task of finding an efficient mapping on the target platform, either based on hardware primitives, where these are available, either emulated in software where these are absent.

2.6.3. Structured Memory Model

As described in Section 2.5.2, NetVM provides a set of orthogonal memory segments that reflect the needs of the programmer for storing temporary or persistent state, and for communicating values between different modules of a packet-processing application. This enables a specific memory location to acquire a semantic meaning for both the programmer and the compiler.

In particular, the presence of an explicitly identifiable memory representing the packet buffer is extremely important, either because several Network Processors (e.g. the Xelerated X11 and the Cavium Octeon) give it a special treatment, either because, as will be pointed out in Section 4.5.1, even on general purpose architectures like the Intel x86, this enhances the opportunity of deploying very effective special purpose optimization techniques.

On the other hand, the flat memory model employed in other programming models like the one of the traditional C language, besides preventing the deployment of special purpose optimizations specifically based on the actual meaning of a memory buffer (e.g. the buffer containing packet data), it cannot be mapped on some NPU architectures, like those of the Intel IXA family, or the Xelerated X11, which are based on an explicit hierarchy of memories. The solution commonly employed in such cases is to extend the language, introducing special storage classes for informing the compiler about which memory of the target architecture should be used for containing a user buffer. For example, the Intel IXP2xxx network processors provide separate interfaces for accessing SRAM and SDRAM memories, which are characterized by different latencies, costs and sizes. Besides, each processing element of the NPU (called "Microengine") owns a small and fast private memory called "scratchpad". The programmer is in charge of deciding in which of these memories should reside each specific portion of the state (e.g. usually packet data is stored in SDRAM, while single static values like counters are stored in the scratchpad memory), so the Intel IXA SDK provides a programming language derived by C (Microengine C) [27], which has been extended with a set of architecture-specific storage classes for allowing to specify at which kind of memory a pointer should refer. Figure 5 shows an example of such scenario, where the packet buffer is mapped on SDRAM. The storage class of a pointer is specified through the `__declspec` keyword.

It appears obvious that such kind of solutions, which make visible the characteristics of the target platform to the programmer, pose a strong limit to portability. On the other side, NetVM memories reflect the purpose for which the programmer use them, i.e. (*i*) accessing the packet buffer (packet memory), (*ii*) communicating values between consecutive modules (info partition), and (iii) storing persistent and static data (data memory). No information is given to the programmer about which kind of memory will be actually used for mapping them on the target architecture, since such task is completely left to the compiler, which can always chose the more efficient solution with the result of enabling portability while still ensuring efficiency.

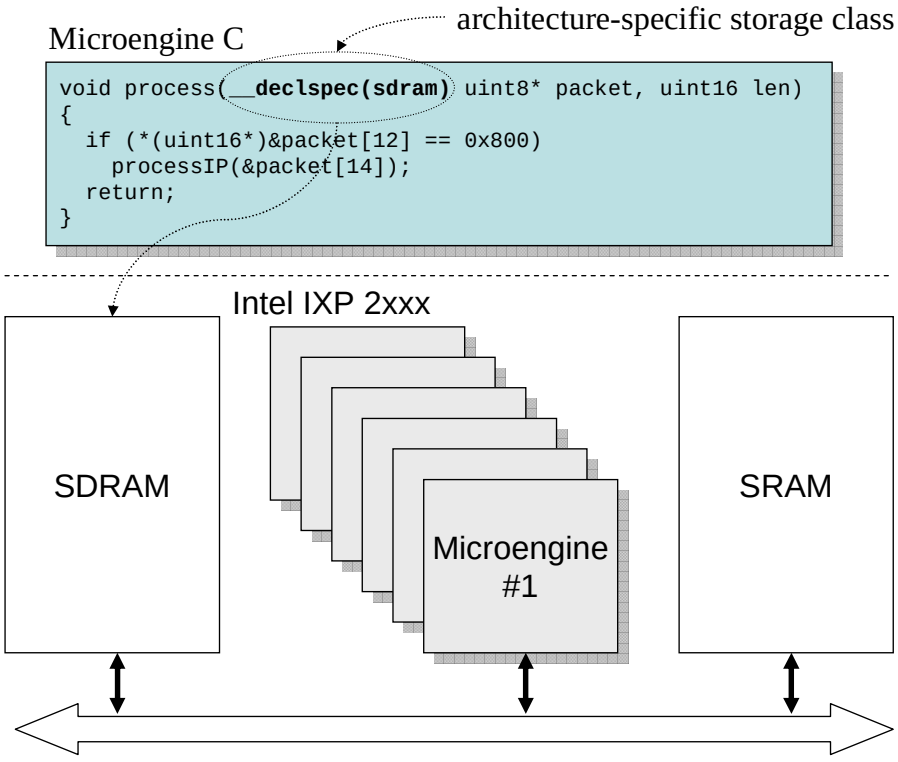


Figure 5. Use of architecture-specific storage-classes for mapping Intel IXP2xxx memories in C

2.6.4. Virtual Coprocessors

Another aspect that is very critical for efficiently mapping packet processing applications on network processor architectures is the exploitation of advanced

functionalities that may be implemented in hardware as coprocessors (e.g. hashing, lookup, string-matching, etc.).

The available solutions can mainly be ascribed to one of the followings: (*i*) encapsulating advanced features in function libraries, (*ii*) exposing them as intrinsics or compiler known functions, (*iii*) using inline assembly. Unfortunately, each one of these methods suffers from the point of view of portability. First, libraries are based on the concept of function call, which is not always available in all network processor architectures, e.g. those of the Intel IXP family, or on systolic array network processors like the Xelerated X11 [4]. Moreover, libraries are software components that are compiled and optimized separately, thus preventing aggressive application-wide optimizations. Intrinsics may represent a solution, because the compiler know their semantic and can map them efficiently on the available hardware features, however, when they abstract low level functionalities, the result is source code highly tied to the target architecture. Finally, inline assembly provides a high potential from the point of view of efficiency, but it highly prevents portability, as well as maintainability and dependability.

The solution proposed by NetVM virtual coprocessors can be viewed as an extension of the concept of “compiler known functions”. In fact, virtual coprocessors are more like “compiler known objects”, i.e. modules with their own state and with “methods” that provide complex functionalities whose operation is specified by the NetVM model, and which a backend compiler can map in the most efficient way on the target architecture. In particular they can be implemented by leveraging the presence of special purpose hardware, if present, or emulated in software otherwise.

Even though it would be possible for a NetVM virtual coprocessor to abstract a real hardware coprocessor, this would lead to similar problems from the point of view of portability, as those pointed out before. Indeed, virtual coprocessors, instead of

abstracting specific hardware functionalities, they abstract “macro-functionalities” that are commonly employed in packet-processing applications, e.g. exact-match lookup, string matching and regular expression matching, enabling the portability of applications across heterogeneous architectures. In particular, for example the NetVM lookup coprocessor could be implemented using a T-CAM on some architectures (e.g. the Xelerated X11), or as a hash table, possibly leveraging a hashing coprocessor like the one provided by the Intel IXP2xxx network processors, or finally it could be implemented completely in software (e.g. through a binary search tree) on general purpose platforms where no specific hardware acceleration is present, as depicted in Figure 6.

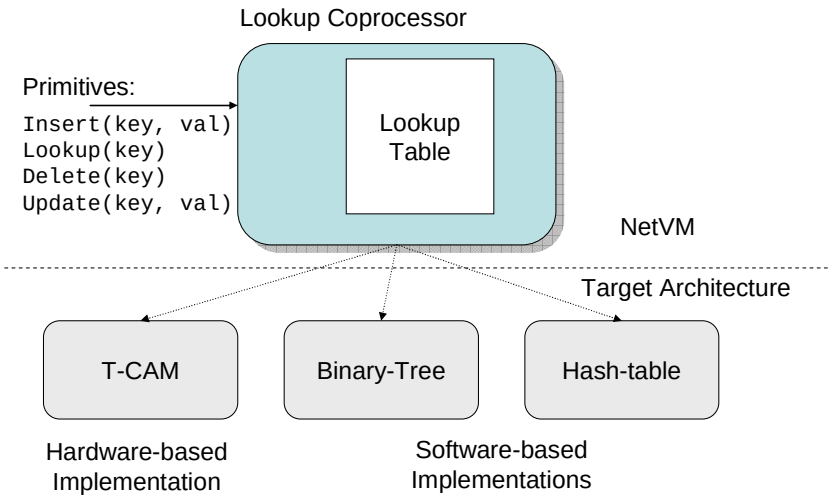


Figure 6. Possible mappings for a lookup coprocessor

2.7. Conclusion

This chapter highlights the need for a suitable abstraction for programming high-speed packet processing applications, capable of allowing them to be both efficient and portable across a wide range of special purpose architectures. The NetVM virtual machine is introduced as a possible solution, showing that portability and efficiency

might not be considered as conflicting requirements. In particular, the NetVM programming model, by capturing the peculiar characteristics of the network processing application domain, provides the programmer with an abstraction layer capable of completely hiding the details of the actual execution platform, thus enabling portability. On the other hand, it also allows a compiler implementing it to have a more detailed view on the semantic of the application, thus enabling the deployment of special purpose optimization and mapping techniques, which favour runtime efficiency.

The goodness of such approach will be demonstrated in the second part of this thesis, where the implementation of the NetVM model in a multi-target optimizing compiler is described, and performance evaluation results are presented.

3. Decoupling Programs from the Knowledge of Protocol Formats

3.1. Enabling Protocol-Agnostic Packet Processing Applications

Packet processing applications such as routers, firewalls and IDSs, rely on protocol demultiplexing functionalities for determining the presence of particular protocol headers in packets, and for extracting the actual values of specific fields to be used for performing more complicated operations. For example, the forwarding process in a router needs to analyze the value of the destination address contained in IP packets for determining the next hop, while a firewall or an ACL module needs to know the values of a given set of fields for performing packet classification.

The traditional approach of hardcoding the format of protocol headers in the software of the abovementioned type of applications, although being efficient in terms of runtime performances, suffers from non-negligible limitations with respect to flexibility and maintainability. In particular, developers must have a deep knowledge of protocol

header format, 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.

An effective way to overcome such problems would be to isolate the knowledge about the format of network protocols in a separate module, by using an application independent language for describing the binary layout of protocols, and by creating a common database of protocol descriptions, usable by several heterogeneous applications. This is the case of the Network Protocol Description Language (NetPDL) [28], formerly proposed by the NetGroup at Politecnico di Torino, which aims at describing the format of network protocol headers and encapsulation rules between different protocols. An API provides the appropriate functionalities for interacting with the protocol description database, allowing user programs to be completely unaware of the exact location of header fields in network packets, and delegating to an external module the task of demultiplexing the headers present in a packet buffer and extracting the actual values of specific fields, as shown in Figure 7.

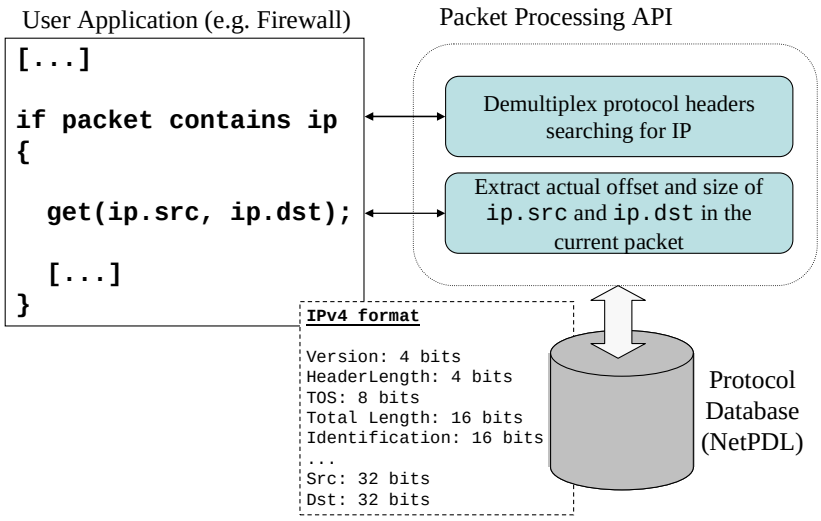


Figure 7. Decoupling applications from the knowledge of protocol formats

However, if the depicted scenario introduces a high flexibility, given by the possibility of seamlessly adding support to novel protocols without any modification to applications, its applicability in the implementation of high-speed data plane network devices highly depends on its capability to compete with the runtime performances provided by the hardcoded approach.

Indeed, NetPDL has been profitably used for implementing a packet-decoder [28], i.e. an engine for parsing the content of network packets and extracting the actual values of each field, according to the information provided by an external protocol description database. Such module is now part of the NetBee library [29] and it is used for visualizing packet-data in the Analyzer [30] network monitor. However, the packet-decoder is based on a step-by-step interpretation of the NetPDL database, and even though its performances can be reasonable for an offline application such as a network sniffer, they are not compatible with the requirements of high speed data-plane applications, such as routers or firewalls, which have to cope with ever increasing line rates.

A solution capable of guaranteeing performances that are comparable to those of completely hand-written programs, consists in translating protocol descriptions into

native code through a compiler. As shown in Figure 8., the NetPDL language could be translated almost one by one into a C or C++ module with the same capabilities of the one based on NetPDL interpretation. However, the use of static compilation would highly mitigate the advantages of having an external and possibly dynamically updatable database of protocol descriptions, because the packet processing module generated from descriptions would suffer from similar problems of its hardcoded counterpart. Indeed, adding support for a new protocol would require extending the external protocol database, regenerating the module and linking it against the user application. Moreover, such scheme would prevent to optimize and tune the generated code based on the needs of the user. For instance, in the depicted scenario, even if the user application only requests the extraction of the ip.src and ip.dst fields, the packet decoding module statically generated from NetPDL would contain code for extracting the values of all the fields of the IP protocol, with a resulting lack of efficiency.

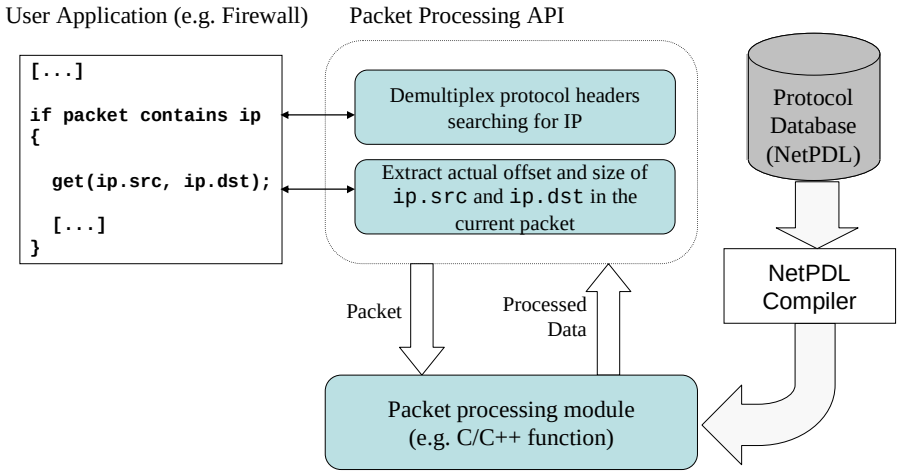


Figure 8. Generation of a packet processing module from protocol descriptions

The second major argument of this thesis is that in order to support an efficient decoupling of the logic of packet processing applications from the knowledge of the format of network protocols, dynamic compilation techniques must be put in place, for generating code to be executed on a configurable packet processor, thus enabling the

dynamic update of the protocol database and the deployment of any suitable optimization.

The solution proposed here relies on an additional language for defining packet filtering and field extraction rules (NetPFL), and on a compiler for translating such rules into a packet processing program for the NetVM, according to the information on protocol format and encapsulation contained in a NetPDL database.

The overall architecture is outlined in Figure 9. NetPFL provides an interface based on simple packet processing primitives that allow shaping NetPDL packet decoding functionalities based on the actual user needs (e.g. specifying the information to be extracted from network packets). Besides, since the operation of the NetVM-based packet processor can be dinamically configured by simply changing the program to be executed, the proposed solution enables a high degree of flexibility, given by the possibility to adding support to new protocols by updating protocol descriptions at runtime. The just-in-time compilation capabilities of NetVM guarantee the runtime efficiency of the approach.

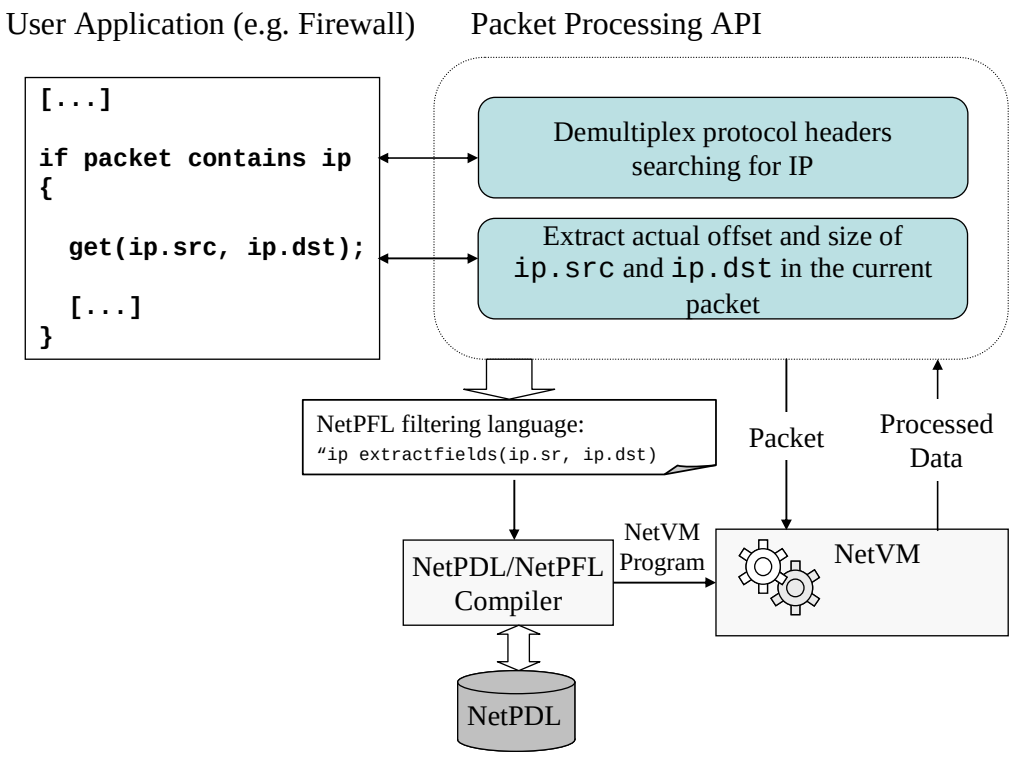


Figure 9. Complete view of the proposed packet processing architecture

As a proof of concept, an optimizing compiler for the translation of NetPDL-based packet filtering rules into a program for the Network Virtual Machine has been designed and implemented, demonstrating that NetPDL can be effectively used for driving the dynamic generation of efficient packet processing programs.

The rest of this Chapter will give an overview on the main building blocks, namely NetPDL and NetPFL (NetVM has been presented in Chapter 2), while the compiler architecture, and the deployed code generation techniques will be discussed in the second part of this thesis.

3.2. Related Technologies: NetPDL and NetPFL

3.2.1. NetPDL

The NetPDL language enables the description of how protocol headers are laid out and chained together inside network packets. Since it is based on XML, specific tags, characterized by several attributes and organized in hierarchical structures, identify the elements of the language.

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 10 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, token delimited and variable size fields, bitfields, padding and more) and by using 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.

While such features are sufficient for the description of L2-L4 protocols, in order to support the description and the recognition of L7 protocols, NetPDL provides advanced features that will be briefly outlined here. More details on the NetPDL language can be found in [31].

```
<protocol name="Ethernet" longname="Ethernet 802.3“>
  <format>
      <fields>
        <field type="fixed" name="dst" longname="MAC Dest." size="6"/>
        <field type="fixed" name="src" longname="MAC Source" size="6"/>
        <field type="fixed" name="type" longname="Ethertype" size="2"/>
      </fields>
  </format>

  <encapsulation>
    <switch expr="buf2int(type)">
      <case value="0x0800"> <nextproto name="#IP"/> </case>
      <case value="0x0806"> <nextproto name="#ARP"/> </case>
    </switch>
  </encapsulation>

</protocol>
```

Figure 10. NetPDL description of the Ethernet protocol header

3.2.1.1. Protocol verification

TCP/IP has an ambiguous mechanism for application-layer de-multiplexing. For instance, while a value 0x800 in the ethertype field uniquely identifies an IP packet, the value “80” in the TCP port does not necessary mean that the packet contains an HTTP payload. For instance, several peer to peer application use this port using custom protocols other than HTTP. In order to allow some form of validity check on the protocol to guarantee that the packet really is what it appears to be, NetPDL provides the `<verify>` construct, which includes both an expression and a set of associated actions. The verification can either return “`found`” or “`not found`”, or it can postpone the result with a “`deferred`” or “`candidate`” return code. The “`deferred`” is used for protocols that require several packets to be analyzed in order to return an exact answer (e.g. RTP, Skype). Vice versa, the “`candidate`” is used for protocols in which the payload can match several protocols at the same time. For instance, KAZAA communicates through HTTP messages that contain a special optional header; hence KAZAA packets are also valid HTTP ones. However, NetPDL is able to differentiate among these protocols and pick the correct one (in this case, the check against the HTTP signature returns “`candidate`”, and this protocol will be the

correct one unless a check against another protocol returns "`found`", in which case the second protocol is chosen).

3.2.1.2. Session Tracking

Session Tracking is mostly used to keep track of TCP sessions. This mechanism leverages a simple table containing the 5-tuple that includes the ID of known sessions and the associated application-layer protocol.

In order to implement session tracking, NetPDL defines a special bi-dimensional variable, the `<lookuptable>` element, which supports an arbitrary number of fields. Fields are either keys to locate entries ("primary key" in database terminology) or data (such as protocol ID) related to the given element.

Although bi-dimensional variables can have any use, they are particularly useful for transport-layer session tracking. These entries (e.g. TCP sessions) have the necessity of being properly managed, e.g., we must be able to purge "zombie" TCP sessions that are no longer active. For this reason, NetPDL can associate an attribute to each entry, defining its validity. An entry can last forever (unless deleted by an explicit command in the NetPDL file), or it can be automatically cleared off after a given inactivity time.

3.2.1.3. Support to application-negotiated sessions

For the case of applications that dynamically negotiate the parameters of the session, e.g., the case of FTP data connection whose ports are dynamically negotiated in the FTP control channel, or SIP sessions that dynamically negotiate RTP ports, NetPDL supports a set of processing elements through the `<execute-code>` section. For instance, the definition of the FTP protocol will contain a piece of code that recognizes the negotiation of a new FTP data session, and inserts a new entry into the TCP session table. Usually these entries do not have to go through a verification process – i.e., if the

"master" session is trusted (it has already been verified before), its "child" sessions should be trusted as well.

3.2.2. Defining actions: NetPFL

Even though 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 direct means for defining actions to be executed when specific conditions are satisfied. Here is where the Network Packet Filtering Language (NetPFL) [32] comes into play.

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. The filtering expression 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). In NetPFL, basic predicates can be composed with the Boolean operators AND, OR, and NOT in order to express complex filters. Since the filtering expression is an optional part of a NetPFL statement, when a filter is not specified, the action should be applied to all incoming packets.

Figure 11 shows two sample NetPFL rules: the first represents a complex filtering expression based on the presence of a `tcp` header and on a condition on the `ip.src` field, while the second is a field extraction statement for returning the values of the `ip.src`, `ip.dst`, `udp.sport` and `udp.dport` fields contained in each `udp` packet.

```
ip.src == 10.0.0.1 and tcp returnpacket as stream 1
udp extractfields(ip.src, ip.dst, udp.sport, udp.dport) as stream 2
```

Figure 11. NetPFL expression examples.

NetPFL is built on top of NetPDL and its main tokens (i.e. protocol names and header fields) are not specified explicitly in the language, but are defined in a NetPDL

database. In other words, the expressions in Figure 11 make sense only if the NetPDL description contains the definition of the specified protocols and fields, i.e. a protocol named "`ip`" whose header contains the fields named "`src`" and "`dst`", a protocol named "`tcp`", and a protocol named "`udp`" with two fields named respectively "`sport`" and "`dport`".

For a detailed specification of the NetPFL language, please refer to [32].

3.3. Conclusion

This Chapter outlines the architecture of a possible solution for efficiently decoupling the logic of packet processing applications from the knowledge of the format of network protocols, and presents its main building blocks.

Using the NetPFL language, a user application can specify the kind of information to be extracted from network packets, while the actual format of supported protocols resides in an external NetPDL database of protocol descriptions. NetPFL rules are used for driving the translation of NetPDL descriptions into code to be executed on the NetVM virtual machine, which can be compiled just-in-time in order to guarantee runtime performances.

This approach enables both flexibility and efficiency, overcoming the limitations either of an approach based on interpretation, either of an approach based on the static compilation of NetPDL descriptions.

The validation of such solution is presented in the second part of this thesis, also reporting performance evaluation results.

PART II. Validation

4. Implementing the NetVM Model

4.1. Introduction

In order to demonstrate the goodness of the NetVM programming model and its capability to enable the creation of portable and efficient packet processing software, the NetVM architecture has been implemented as a portable runtime environment and a multi-target optimizing compiler infrastructure. The compiler is able to operate either as a Just in Time or as an Ahead of Time compiler, generating native or assembly code, depending on the target platform. Optimizations work on two different levels: the higher level is architecture-independent and operates on the code removing redundancies and useless computations, whereas the lower one is target-specific and performs the actual mappings between the NetVM model and the target machine, possibly exploiting special purpose hardware units available on modern NPUs.

Experimental results reported in Chapter 7, demonstrate the effectiveness of the approach, showing that thanks to the characteristics exposed by the NetVM model, the generated code has performances often better than those obtained from hand-written programs compiled with state-of-the-art general-purpose compilers.

4.2. The NetVM Framework

The NetVM model requires a runtime environment acting as a communication layer with the external world. Its main function is to provide I/O facilities, to handle the coprocessors implementation (hardware or software) and to manage the application's resources, e.g. memory allocation. In fact a NetVM application needs to receive packets from input interfaces and to forward them to output interfaces after the processing. Such operations are heavily dependent on the hardware characteristics. In other words, the runtime environment must implement an abstraction layer making all such details transparent to the application and to the programmer.

On the other hand, since a NetVM application relies on different elements (NetPEs, coprocessors, etc), whose configuration can be chosen by the programmer, the runtime environment has to (1) allow the programmer to create and configure each component, and (2) implement these elements on different architectures either by exploiting hardware modules or by supplying software implementation of unavailable components.

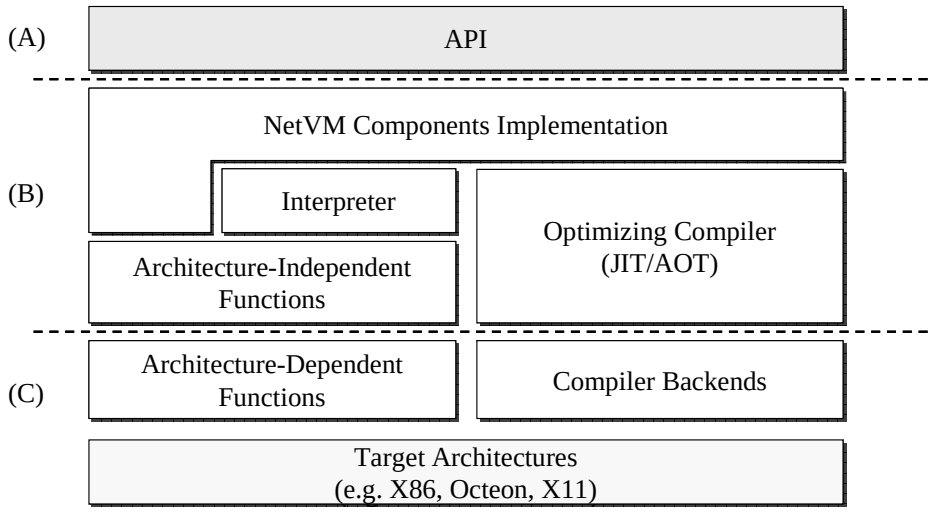


Figure 12. NetVM Framework Architecture

The NetVM model is implemented as a framework, (whose logical layout is shown in Figure 12), which comprises a portable runtime environment and a multi-target

optimizing compiler. At the top of the framework (A) sits a programming interface that allows the programmer to instantiate and manage the main NetVM components in the user applications. The middle layer (B) represents the core of the framework, implementing the architecture-independent parts of the runtime environment and a NetIL interpreter, as well as the target-independent components of the compiler. Finally, at the bottom of the structure (C) we find target specific modules, i.e. the compiler back-ends and the architecture-specific parts of the runtime environment, which implement the actual mapping of the NetVM functionalities (i.e. instruction set and virtual coprocessors), on the target architecture.

4.3. Compiler Infrastructure

As Figure 13 shows, the compiler follows a classical 3-stage model. First, the compiler front-end builds a medium-level intermediate representation (MIR) of the source program, while checking its formal correctness; then the MIR is fed into the optimizer, whose objective is to reduce code redundancies and improve efficiency. A platform-dependent back-end lowers the optimized MIR to a low level intermediate representation (LIR), which is very close to the assembly language of the target architecture and performs additional optimizations. Finally, the resulting machine code is emitted.

A program represented in MIR form is described as a list of expression trees, whose root nodes represent statements (i.e. assignment and control flow operators), while leaf nodes represent the operands of an expression (e.g. constant values or registers). The LIR form, instead, represents the program as a sequence of three-address instructions closer to the target machine language. The reason for implementing a multi-level intermediate representation is based on the need to delay the lowering phase and to provide as much information as possible on the source program to the optimizer. This

makes it possible to perform more aggressive optimizations, based on the knowledge of the semantic of the constructs employed by the programmer, as will be pointed out in Section 4.5.

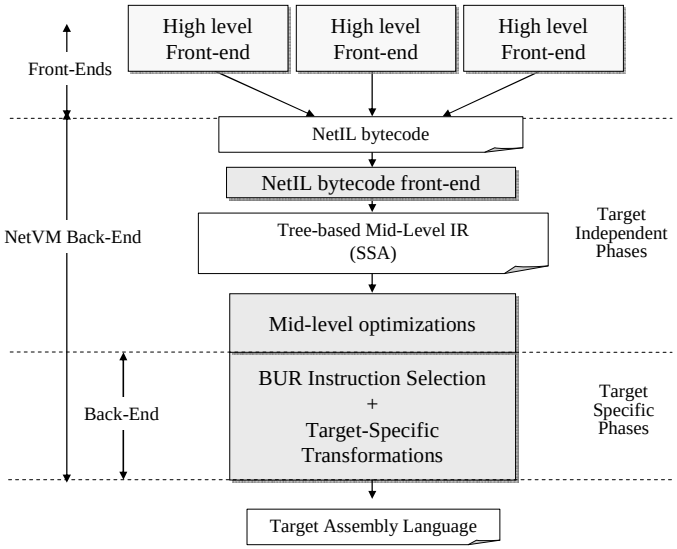


Figure 13. Compiler Architecture

The whole compilation framework is designed in a modular fashion, in order to ease the task of adding new back-ends. In particular, the analysis and optimization algorithms are able to work on different intermediate representations, and each back-end can configure the optimizer in order to apply only the transformations that are suitable for the target platform.

The compiler can generate either machine code in memory, following the Just-In-Time paradigm, or assembly files as an Ahead-Of-Time compiler. In the latter case, the programs generated by the compiler are assembled by using third party tools (e.g. GCC or the development tools provided for the specific target platform).

4.4. The Compilation Flow

Although a source program can be translated directly into the target language, compilers are generally organized as a series of phases, each of which apply a distinct transformation

to the source program. This scheme creates the need for an intermediate representation for the code that is continuously transformed during the compilation process. Since the details of the target language should be confined to the compiler backend as far as possible, the use of a target-independent intermediate provides the following benefits:

- Retargeting is facilitated: a compiler for a different target architecture can be created by only creating a new back-end
- All target-independent optimizations can be applied to the intermediate representation before passing it to the backend

The NetVM compiler uses two different representations for the program being compiled: a Medium Intermediate Representation *MIR* and a Low Level Intermediate Representation *LLIR*. The former is a machine-independent representation created by the compiler front-end and it is transformed into the latter, machine-dependent, by the instruction selection phase of every backend.

In *MIR* form, the code is described as a list of *statements*[1]; each statement represents a tree whose nodes represent expressions. The operators employed in this phase are NetIL ones, allowing the compiler to exploit the knowledge of the semantics of domain-specific constructs exposed by the language, as pointed out in Section 2.6. The operand stack is mapped on expression trees, while operations on local variables are converted into operations on an infinite set of registers, called “virtual registers”. Figure 14 shows an example of a NetIL program being converted into a list of MIR statements.

[1]Statements are roughly equivalent to sentences in natural languages. A statement forms a complete unit of execution: `a=b+c` is a *statement*, while `b+c` is an *expression*.

The *LLIR* intermediate representation is a list of assembly instructions whose operators are very close to the target machine language. Each backend maps MIR statements on lists of LLIR instructions and then applies on it target-specific transformations and optimizations.

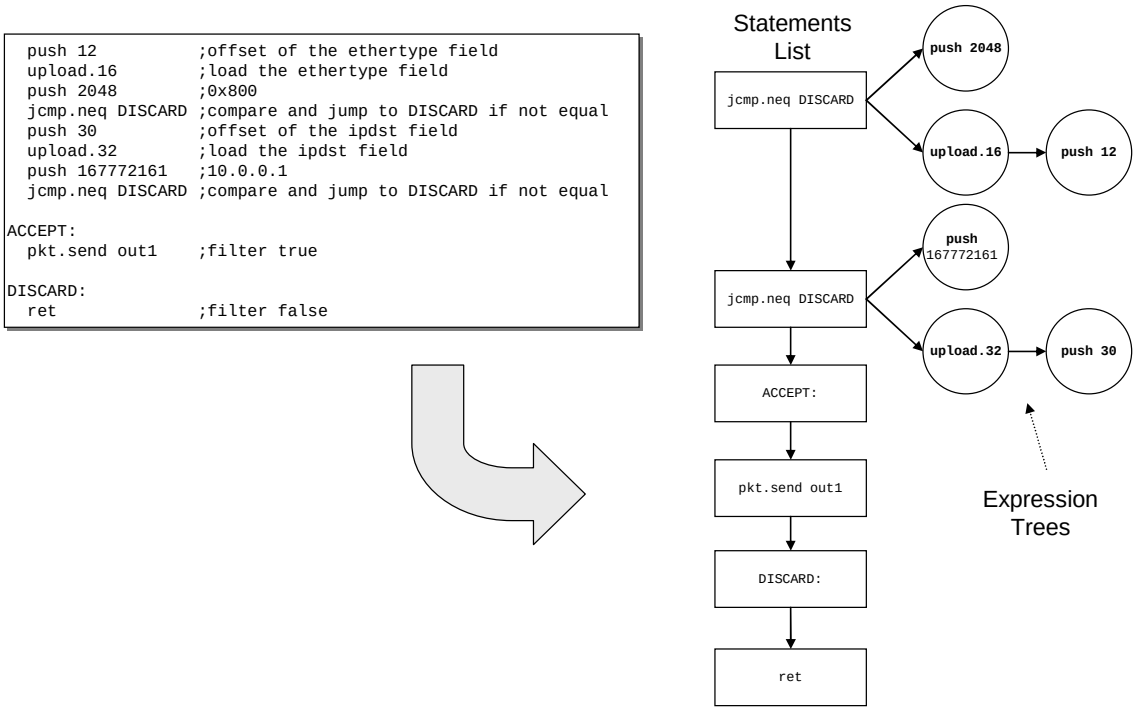


Figure 14. Conversion of a NetIL program into a MIR list of statements

Figure 15 shows an overview of the compilation flow regarding mid-level transformations, which will be the argument of the rest of this section.

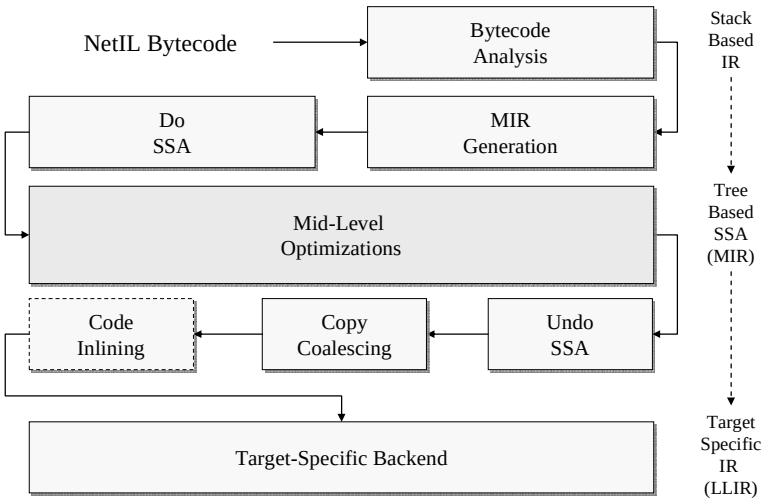


Figure 15. Compilation Phases

4.4.1. Mid-Level Optimizations

In order to provide a general framework for simplifying the development of dataflow analysis and optimization algorithms, the NetVM compiler translates the MIR into a Static Single Assignment form (SSA) [33]. The SSA form implies that every variable is assigned exactly once, in this way the relationships between the definition and the uses of every variable are made explicit in the MIR, without altering the semantics of the program. The optimizing algorithms benefit from this form in terms of simplicity of implementation.

The optimization algorithms implemented in the NetVM framework have been selected after an accurate analysis of existing NetIL code, either hand-written, or automatically generated through a set of high-level frontends. In particular, the code generated by the packet filter compiler presented in [25], exposes several redundancies and suboptimal recurrent patterns. The implemented algorithms aim also at taking into account such situations, by removing the negative effects introduced by automatic code generation.

Among the implemented optimization algorithms, Constant Propagation replaces every use of constant-initialized registers with the respective values. Such optimization

removes assignment instructions where a constant is copied into a register whose value is never changed and often enables the application of other optimizations, such as Constant Folding or Dead Code Elimination. The former of these tries to simplify all the operations whose operands are constant, by replacing them with the result computed at compile-time. The latter removes instructions defining variables that are no longer used later in the code (i.e. dead variables). Algebraic Simplification has some similarities with constant folding, but, instead of computing at compile time the result of constant expressions, it exploits algebraic properties of mathematical and logic instructions to replace sub-expressions that can be computed at compile time with their result, for example by substituting the expression (*a * 1*) with (*a*). Reassociation is a technique that joins different statement trees into deeper ones, enabling further transformations to be applied by other algorithms like Constant Folding [34].

The role of reassociation is evident when considering the structure of typical packet demultiplexing programs. These programs usually contain sequences of operations for finding the offsets of both protocol headers and fields in the packet buffer. Figure 16A shows an example of such a sequence of statements for incrementing a variable holding the current offset (i.e. r0), in order to point to the beginning of the TCP header. The increment is made in two steps, by adding the lengths of the Ethernet and IP headers (14 and 20 bytes respectively). The reassociation algorithm joins the two statements resulting in the statement on the left of Figure 16B, allowing further optimizations. Indeed, constant folding can remove the second ADD node, resulting in the tree on the right. Since this kind of pattern is very frequent, reassociation is very effective in terms of performance gain.

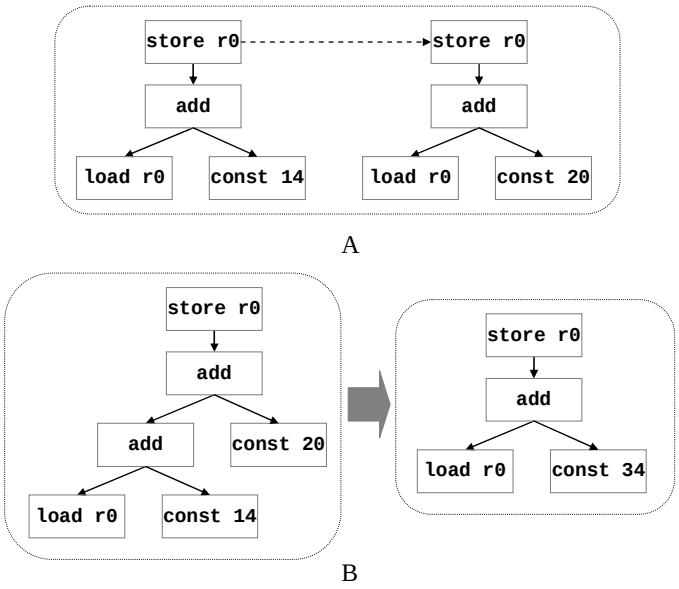


Figure 16. The role of reassociation

All optimizations described above are performed on the IR in SSA form, but in order to produce executable code, this has to be reverted back to a normal form: this step leaves the program in a state where most variables are defined only once and a large number of copies exist in the program. This is clearly non-optimal because such quantity of copies is cumbersome to execute and a large number of virtual register can burden subsequent compiler modules, affecting compilation times. For these reasons we implemented a Copy Coalescing [35] algorithm, which scans the code for copies and tries to assign the same name to both the source and the destination variables involved in the copy. This is safe if the variables involved have live ranges that do not overlap. Beside optimizations based on dataflow analyses, the optimizer also provides algorithms for simplifying the structure of the control flow graph, such as Branch Simplification, for replacing all conditional jumps that can be evaluated at compile-time with unconditional jumps, Jump-to-Jump Elimination for bypassing and removing basic blocks containing only a jump instruction, and Unreachable Code Elimination for removing unreachable basic blocks [34].

Although the architecture-independent optimization algorithms implemented look simple and are widely known from classical compiler theory, they have proven to be extremely effective for two main reasons: (*i*) packet-processing applications use a very simple structure of the code, compared to general purpose ones, and (*ii*) these provide the base for further target-specific transformations that can be applied by a specific back-end, as will be detailed in 4.5. The combination of both architecture-independent and target-specific optimizations results in the production of code that in some cases is faster than the one generated by state-of-the-art C compilers, as Chapter 0 will show.

4.5. Compiler Backends

The NetVM compiler infrastructure provides three backends: one for the Intel x86 architecture, one for the Cavium Octeon network processor and one for the Xelerated X11 systolic array processor. In particular, the former two have a very similar structure, while the latter, being targeted to a very special purpose architecture, relies on a more complicated sequence of compilation phases.

Every backend of the NetVM compiler translates MIR statements into sequences of LLIR instructions implementing them. This task is handled through a Bottom-Up Rewriting System (BURS) [36], which executes a tree-matching algorithm driven by architecture-specific rules that specify how a portion of the intermediate representation (i.e. an expression sub-tree) should be translated into target instructions. In particular, different rules can relate to overlapping tree patterns, and the BURS is able to chose the best (i.e. the less expensive) combination that covers the most extended expression tree. BURS can be configured to recognize very specific patterns that can be part of an algorithm, enabling a very flexible approach in the creation of the target code. For instance, an algorithm made up of three pieces ABC can be implemented as AB in

software and C in hardware on one platform, and as A in software and BC in hardware on another platform.

4.5.1. X86 Backend

The x86 backend follows the Just-In-Time paradigm: for each NetPE composing a NetVM application it generates the binary code for a function receiving an Exchange Buffer as an argument. The sequence of the compilation phases involved is shown in Figure 17.

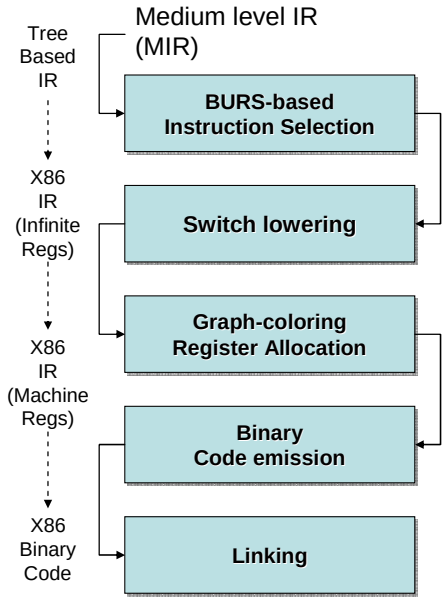


Figure 17. Compilation phases for the x86 backend

The x86 backend, after having mapped MIR statements onto x86 LLIR instructions in the BURS instruction selection phase, performs register allocation in order to assign a machine register or a memory location to every virtual register used in the MIR program. The register allocation algorithm implemented performs is based on graph coloring [37][38], using the spill heuristic proposed in [39] for minimizing spill costs and for guaranteeing an optimal utilization of machine registers.

4.5.1.1. Intel X86 low-level optimizations

The set of BURS rules implemented in the back-end aims at addressing two problems: (*i*) the optimal exploitation of the complex instruction set of the target machine, and (*ii*) the application of specific optimizations for packet-processing applications.

With respect to the first kind of optimization, the CISC-based Intel x86 includes powerful and complex instructions, which allow specific NetIL patterns to be translated into a single x86 instruction, with the result of minimizing the code size. The BURS instruction selection algorithm makes this operation straightforward. For example, Figure 18 presents a fragment of x86 code that calculates the length of the IP options fields with both its naïve and its optimized version. Since this value is calculated by loading the IP header field, masking it, multiplying it by four and finally subtracting 20, we can compact most of the processing through the x86 LEA (Load Effective Address) instruction [40], which exploits the Memory Management Unit of the processor.

Non optimized	Optimized

`movzx eax, byte ptr [ebx+14]`
`and eax, 0xf`
`mov esi, 4`
`mul esi`
`mov esi, eax`
`add esi, -20`
	`movzx eax, byte ptr [ebx+14]`
	`and eax, 0xf`
	`lea ecx, dword ptr[ecx+eax*4–20]`

Figure 18. Exploiting the Intel x86 instruction set

On the other hand, we implemented special rules for optimizing frequent operations of packet-processing applications. For example, these often need to load a field from the packet header, perform some calculation and compare it with a constant value. However packets contain data organized in network byte order, which is big-endian, while x86 uses the little-endian convention. This requires swapping the data contained in the packet buffer before starting the processing. Our solution, instead, uses the BURS to recognize those patterns of instructions and move the byte swapping operation to

compile time. In other words, whenever possible, instead of generating code for swapping the bytes of a register at runtime, the compiler swaps the constant during the compilation, thus producing more efficient code. A simple example of the use of this technique is presented in Figure 19, which refers to the control that determines if an Ethernet header is followed by an IP header.

Non optimized	Optimized

`mov eax, word ptr [12]`
`shr eax, 0x10`
`bswap eax`
`cmp eax, 0x800`
	`cmp word ptr [12], 0x8`

Figure 19. Constant byte order swapping optimization

Another common operation in packet-processing applications is represented by the multi-way branch, modelled after the switch-case construct of the C language. The back-end includes a switch lowering module that follows an approach similar to the one implemented in the LLVM compiler [41], which is able to select the best mapping algorithm, according to the cardinality and the density of the case set.

Finally, the x86 back-end includes a specific phase that implements an efficient linking strategy for code associated to different NetPEs: direct linking avoids returning the control to the framework when a NetPE task ends, hence reducing the overhead introduced by the runtime environment.

4.5.2. Octeon Back-end

Before describing the backend for the Cavium Octeon network processor, a short description of the characteristics of the target architecture is presented, and more details on it are reported in Appendix A.2.

4.5.2.1. The Octeon architecture

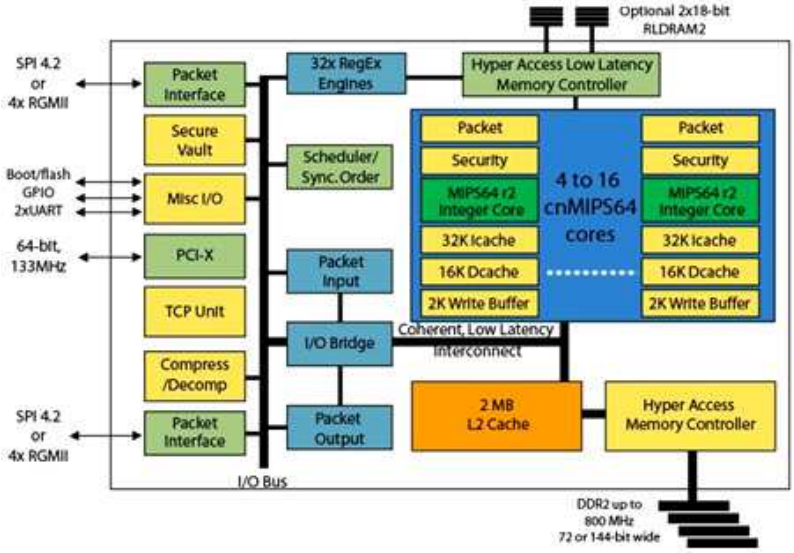


Figure 20. Internal Architecture of the Cavium Octeon Network Processor[2]

Like most NPs, the Cavium Octeon tries to exploit the parallelism of typical packet-processing applications: for this reason it features up to 16 MIPS-64 cores at 600 MHz. Each core has a private L1 cache, while the L2 cache and DRAM are shared. Although the L2 cache and DRAM are physically shared, the cores cannot communicate through the memory because of their private virtual memory space. Communication primitives between cores are provided by specific hardware mechanisms. The primary on-chip communication mechanism is the work, which is an entity created upon the arrival of a packet and queued into a specific hardware unit: the Packet Order Work (POW). Works have many attributes that determine how the POW schedules them to the cores. For example the programmer can specify different QoS levels associated with different

[2] Copyright © 2000 - 2008 Cavium Networks. All rights reserved (http://www.caviumnetworks.com/OCTEON-Plus_CN58XX.html)

kinds of traffic, since the unit receiving incoming packets can parse the packet header, providing a preliminary classification. The most important attribute is the group: in fact cores subscribe to groups and the POW schedules works to the cores according to the subscribed groups. When a core terminates its job, it can submit the work to another group, i.e. to another core, or send the packet out to a network interface.

Besides the MIPS cores, the chip also contains supporting units and coprocessors for offloading some specific tasks. In particular, some of these deal with the reception and the transmission of packets, others are devoted to the management of pools of memory buffers, while coprocessors implement cryptographic and string matching functionalities in hardware.

4.5.2.2. The compiler back-end for the Cavium Octeon

When generating code for the Cavium Octeon, the NetVM compiler uses an Ahead-Of-Time model and the output of the compilation process is represented by several assembly files, C listings and configuration files that must be further processed by the Octeon SDK, using the well known GCC compiler toolchain. The result is a native application running on the NP hardware with a minimal runtime environment, as the processor units are exploited to implement natively the NetVM model. In fact, as figure Figure 21 shows, the code generation process is not different from the x86 back-end (i.e. it implements the BURS instruction selection and global register allocation), while the mapping of native hardware functionalities deserves some more discussion and represents the most valuable part of this work. Particularly, this includes the mapping of the Exchange Buffer (i.e., the memory that contains the packet) on native hardware structures, and the mapping of the string matching coprocessor of the NetVM model.

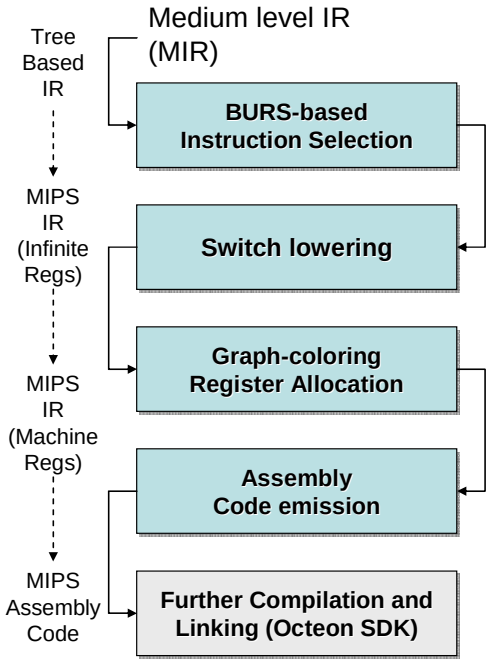


Figure 21. Compilation phases for the Cavium Octeon backend

With respect to the former, the Exchange Buffer can be mapped on the work structure of the POW unit. This enables NetPEs to be distributed on different cores that communicate through the native mechanism, in a way that is completely transparent to the programmer. Currently, our prototype exploits only one core, hence it implements dynamic NetPE linking as in the x86 back-end and exploits the POW unit only for receiving and transmitting packets from the external world. However the general mechanism is already in place and can be used as a starting point for future work aiming at fully exploiting the potentialities of multi-core processing.

With respect to the second item, the NetVM model has a general string matching coprocessor that enables searching for groups of patterns in the packet payload. Patterns, which must be initialized before starting the program, are divided into groups identified with an integer ID, so that the coprocessor can search all the patterns belonging to a group at once and return multiple matching results to the caller. While the x86 back-end provides a software implementation based on the Aho-Corasik algorithm [42], the Octeon includes a hardware unit that is able to traverse graph-based

structures representing Deterministic Finite Automata (DFA) in memory, which can be used to perform both string and regular expression matching. With respect to the Octeon processor, the DFA graph must be translated into a binary image, which has to be loaded in a special external memory, the Low Latency Memory (LLM). During execution, the cores can submit a command to the DFA engine specifying the address of the packet payload and the address of the graph in the Low Latency memory to be used. The hardware unit automatically loads data from the packet memory and uses it to traverse the graph in the LLM, while searching for a match.

Finally, the runtime environment for this back-end is very simple and it consists of an initialization routine (automatically emitted by the compiler) that initializes the processor units and instantiates the memory structure needed by the NetVM instance. The only task of the runtime environment is then to receive packets from interfaces and to pass them to the NetVM.

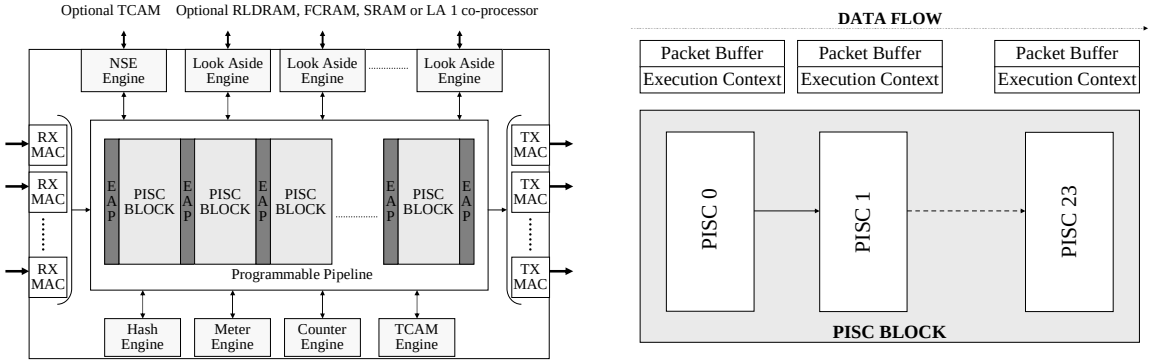
4.5.3. X11 Backend

4.5.3.1. The X11 Network Processor

The Xelerator X11 network processor is based on a systolic array (actually a pipeline) with a synchronous dataflow architecture, which shares the concept of a systolic pipeline with its predecessor X10q [43]. Figure 22 shows an overview on its internal architecture.

The processing elements are either VLIW processors called Packet Instruction Set Computers (PISCs) or I/O processors called Engine Access Points (EAPs). As shown in Figure 22a PISCs are arranged in blocks while EAPs are placed at fixed points between PISC blocks. EAPs essentially dispatch the computation to special purpose devices that can be used to offload part of the computation off the PISC pipeline. Such devices include TCAMs, counters, hardware for computing hash values, external SRAM, etc.

When a packet enters the pipeline, it is first partitioned into fixed size fragments. Thereafter, the pipeline processes the packet fragments using iterations of (1) PISC processing interrupted by (2) actions and look-ups orchestrated by EAPs. As a fragment traverses the pipeline, it carries an individual execution context containing the fragment itself, a register file, status registers, and other information that constitute the complete state of a program. Figure 22b shows the details of a PISC block. It is important to understand that one PISC acts on one packet fragment during exactly one cycle. During this cycle, the PISC can perform a set of parallel instructions on the fragment, before passing it on to the next element in the pipeline.



a) X11 Architecture b) PISC Block Detail

Figure 22. X11 Internal Architecture Overview

The parallelism of the pipeline is hardwired in the architecture itself. From one perspective, this makes the software handling of concurrency easy, since the execution contexts and PISCs are effectively isolated from each other. No explicit mechanisms such as threads or mutexes need to be adopted to protect accesses to these local resources. It is also easy to access external resources as long as this is made in a constrained fashion, primarily limited by the look-up bandwidth towards external engines.

However, generic update of shared state is difficult to realize due to pipeline hazards, including Read-After-Write, Write-After-Write, etc. [44]. The reason is that the non-

stalling nature of the synchronous pipeline makes it impossible for a program to wait indefinitely for an asynchronous mutex. However, for the X11, a mutex mechanism can be achieved by looping or by controlling the traffic scheduling into the systolic pipeline. If no hardware-provided mechanisms exists, all such shared accesses need to be scheduled in advance when configuring the pipeline for a specific application. Fortunately, the X11 architecture offers some means for providing more elaborate accesses to shared resources. This includes support for atomic read-and-increment operations both on the on-circuit counters engine as well as external RAM locations.

From the compiler perspective, a X11 packet program consists of a number of instruction sequences that are laid out in the instruction memory of the pipeline. This memory is actually a two-dimensional matrix with rows and columns where the control flows unidirectionally and synchronously between columns, and branching occurs between rows. Because of the unidirectional execution flow, loops are not possible by definition; branches, however, are allowed. The layout of code in the instruction memory can be seen as a two-dimensional optimization problem, where a vertical column constitutes the instruction space of a single PISC, and the horizontal rows are instruction sequences. The execution context contains a row instruction pointer so that PISCs know which instruction to execute. Branching modifies the row instruction pointer but does not affect the horizontal flow of the program.

The drawbacks to this programming model are tied to its advantages. First, looping is not allowed: programs requiring loops need to be unfolded to some limit that fits the pipeline. The X11 also provides a loopback path to let packets re-enter the pipeline if the program is longer than the number of pipeline stages allows. The number of pipeline passes, k, is statically configured at link-time and is limited since the throughput is proportional to k^{-1}. The operating frequency of the X11 systolic pipeline is dimensioned to allow a specific number of loops while still providing wire-speed.

Moreover, in order to avoid reordering, all packets coming from the same input interface always undergo the same number of pipeline passes, even if the processing could terminate earlier for some of them.

Second, there are few methods to share state between packets. In particular, it is difficult for information from one packet to influence the processing of another. This includes programs that adapt to traffic contents traffic, e.g., stateful packet filters. To provide for shared state between packets, one can use the support from the existing counter engine or implement some other, more elaborate mechanisms in the general-purpose look-aside engines. It is also possible to communicate with the control plane, which in turn can re-program the pipeline by altering the state of look-up tables, but this approach has the obvious drawbacks of being limited in bandwidth and also may introduce race-conditions.

4.5.3.2. A Back-end for the X11 NPU

The architecture of the X11 backend is shown in Figure 23.

The back-end translates the tree-based intermediate representation generated by the upper layers of the compiler into the LLIR, while mapping the accesses to virtual coprocessors on instructions that make use of the special purpose hardware features (e.g. TCAMs) available on the target architecture. This task is performed by the Bottom Up Rewriting System instruction selection phase.

In contrast to traditional processors, the X11 NPU completely lacks the concept of function call; therefore a NetVM application composed of multiple NetPEs must be transformed into a single compilation unit to be laid out as a linear code sequence throughout the PISC pipeline. The X11 back-end compiler addresses this problem by performing an inlining step in the compilation process, where the code belonging to different NetPEs is linked together by replacing inter-module calls with jump

instructions. This inlining operation is possible only if the NetPE interconnection graph is acyclic, however this property is intrinsically ensured by the NetVM model.

Afterwards, the intermediate representation is further optimized by removing redundant instructions that might have been generated during the instruction selection phase, then the resulting code is examined to detect independent instructions that are suitable to be merged in VLIW blocks. At the end of the compilation process, a resulting assembly file is created which can be used as an input for the X11 SDK tools that create the proper binary files for loading and execution.

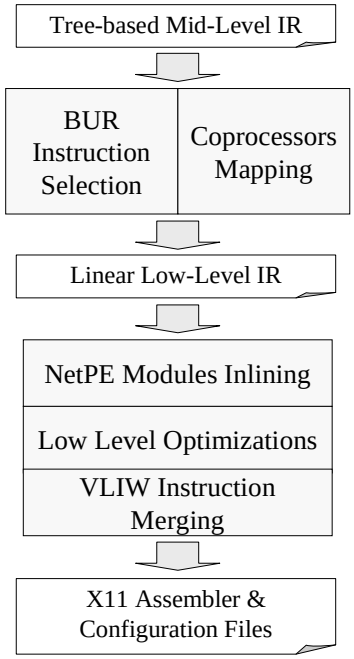


Figure 23. Architecture of the X11 backend

4.5.3.3. The Mapping Process

Compiling a packet processing program for the X11 NPU does not differ significantly from compiling it for any other kind of processor, as long as only the generation of sequences of target instructions from high level constructs is considered. However, some constraints that are specific to systolic architectures, along with some characteristics of the X11 processor, suggest the adoption of specific compilation

techniques in order to best exploit the available hardware resources and to improve the chance of a program to be correctly and efficiently compiled.

This section explores the major problems related to the efficient mapping of NetVM applications on the X11 architecture and presents the most innovative aspects of the NetVM compiler infrastructure.

Handling Loops

Since backward pointing branch instructions are forbidden, systolic array processors are characterized by an "upstream to downstream" execution model, where the control flow is driven by data flowing through the pipeline and cannot be redirected to a previous stage. This translates to the impossibility of mapping generic loops on a systolic array, unless their maximum number of iterations is bounded and known at compile time, so that they can be completely unrolled and laid out as a linear sequence of instructions. However, even in this case some practical problems arise: the theoretical upper bound on the number of iterations may be so large that the resulting overall instruction count could exceed the number of available stages even when using the loopback path as described in Section 4.5.3.1.

If such considerations apparently pose a strong limitation on the kinds of applications that can be successfully and efficiently mapped onto a systolic array network processor, it should be noted that uncontrolled loops are not frequent in standard forwarding programs (either L2 or L3) with the exception of some protocols (e.g., MPLS stacking or IPv6 extension headers [25]). In such cases the problem can be overcome by limiting the maximum number of loop iterations in the source program to a fixed value. Such considerations point out that the theoretical limitation of systolic arrays in handling loops may not be so relevant in practice.

Keeping the State of the Application

The NetVM model uses different memories to keep the state of an application. In particular, state information local to a NetPE is stored in the NetPE local register file and local data memory, the former keeping temporary values while the latter is used for static values as well as complex structures. Vice versa, the state that is local to a packet is stored in the packet buffer and a special buffer called the "info memory", i.e. a memory segment that allows subsequent NetPEs to communicate between them.

On the X11 side, the execution context is represented by the packet memory and a register file, while persistent state must be kept in externally attached memories that are accessed through the EAPs. As a matter of fact there happens to be a significant parallelism between the NetVM model and the X11 processor when it comes to data associated with a packet. In particular, the X11 packet memory and register file allow indirect addressing and can be used to map the NetVM packet buffer and the info memory. Besides, a portion of the register file can be allocated for keeping intermediate results as they are computed in the NetPEs, as well as local register values.

On the other hand, the two platforms differ in the way permanent data (i.e. the state that survives across different packets) is treated. As detailed in Section 4.5.3.1, there are constraints on how multiple, concurrent accesses to the same external memory location can be made. Section 4.5.3.3 reports how in very specific cases the compiler is able to handle this problem while still ensuring the safe update of shared memory locations.

Mapping NetVM Coprocessors

The NetVM model allows complex operations and functionalities to be represented as invocations to virtual coprocessors. The back-end maps them on the corresponding hardware features (if available on the physical device) in order to maximize the efficiency of the resulting code. In particular, on the X11 processor this usually

translates to generating instructions that send and receive data to/from the EAPs and declaring which engine operation should be performed.

A look-up coprocessor that allows the programmer to associate 32-bit keys to 32-bit values was considered as a proof of concept. On the X11 the requested operation can be performed by the integrated TCAM module. Since the same hardware unit must possibly be shared with other instances of the same coprocessor (in a different NetPE) or contain other unrelated content, the compiler provides a thin hardware abstraction layer to split the TCAM into multiple tables. This is achieved by dedicating a portion of the look-up key space to hold a table number.

NetVM coprocessors wrap a well-defined interface around a usually complex algorithm; the compiler has the twofold task of translating the algorithm itself and adapting the interface to the actual hardware units employed. While the latter task is achieved by the compiler, the former might prove impossible due to possible limitations of the hardware platform. In particular, if the target architecture does not provide the specific functionalities exposed by a virtual coprocessor, a software emulation must be performed. However, this might not always be possible due to restricted amount of primitives provided by the hardware and limitations imposed on the instruction count.

Exploiting the Features of the Hardware Architecture

The previous section explored the problem of mapping a virtual coprocessor (i.e., a specialized macro-functionality) defined by the NetVM model on real hardware. This section presents the dual problem, i.e., mapping generic NetIL code to some specialized modules provided by the hardware.

Apart from the case where the source language exposes high level constructs that find a natural mapping on specific hardware functionalities, the problem is in general extremely complex: hardware modules usually implement complex algorithms that, in order to be efficiently translated, must first be recognized in the source program.

The switch-case provides a simple example of an easily recognizable high-level construct. The instruction count of a traditional implementation based on a linear search might grow in complexity with the number of possible destinations, potentially using an extensive portion of the pipeline. However, on the X11 the same behaviour can be obtained by performing an associative look-up that uses the on-board TCAM, costing effectively one pipeline stage only, independently from the number of possible alternatives.

An unintended consequence of extensively using this mapping technique might be the over-subscription of limited hardware resources. In particular, there are limitations in look-up bandwidth and also the fact the EAPs are present at specific stages in the pipeline. In this case the compiler should emit code that uses other pipeline resources such as the PISC processors or different external units. Although deciding when to do this is a complex optimization problem, the compiler tries to solve it through a simple heuristic that works well in the average case.

Making the specialized functionalities provided by the X11 hardware automatically available to the program requires in the general case more effort than mapping the switch-case construct. A good example derives from the problems related to the concurrent update of shared information mentioned in previous paragraphs. If the state to be updated is an integral value, the compiler can make good use of the X11 support for atomic increment instructions, thus becoming able to overcome concurrency issues in a limited set of cases. A common example is keeping counters in external memory, e.g. for statistical purposes.

A counter increment operation in itself is not atomic as it is necessary to fetch the old value, increase it and store the newly computed result at the same offset. However if this procedure is not performed atomically by the hardware it becomes possible for two consecutive packets to read the same value from memory with the net effect of

incrementing the counter once instead of twice. To overcome this issue the compiler uses the BURS-based instruction selector which is able to recognize if specific locations of the data-memory are accessed through this pattern of operations, and to map them on the special purpose atomic increment instructions provided by the hardware.

Depending on how the source code is written, it can happen that a pattern ends up split across different statements. Since the BURS operates on a single IR expression tree at a time, in this case the recognition mechanism does not work. No control on the source code form can be assumed, so this issue would result in low reliability of the compilation process if left unchecked. Vast improvements can be made by processing the intermediate representation with appropriate optimization algorithms, such as algebraic reassociation. These algorithms can rearrange subtrees in the IR so that the semantic meaning of the program is preserved, but providing the instruction selector with deeper trees that are more likely to contain recognizable patterns. This way the BURS can operate successfully even if the related instructions were originally scattered across a region of the source listing.

In any case, it must be pointed out that even though such techniques work well in very specific cases, their general validity still needs to be proven, since they are tuned on patterns of instructions and not on algorithms. In particular, even for the simple example of counters, the programmer could update a specific memory location in several exotic ways, preventing the BURS to recognize the sequence of instructions as a predefined pattern. We believe that in order to deploy a general algorithm recognition technique, more specialized analyses of the code should be performed.

VLIW Instruction Merging

Being VLIW processors, PISCs allow up to four independent operations to be executed at the same time, in order to exploit instruction-level parallelism. These can be (1) an ALU operation, (2) a move for copying words of up to 32 bits between different

locations of the register file and the packet memory, (3) a load offset operation for indirectly accessing the register file or packet data, and (4) a branch.

When generating assembly code, the compiler should try to merge multiple instructions in single VLIW words, taking care appropriately of data and control dependencies. Several algorithms are described in literature for handling such task in an optimal way, e.g., trace scheduling [45]. The compiler currently implements a simple algorithm that works only on straight-line code fragments (i.e., basic blocks) and does not perform any instruction reordering before merging. This provides good results, even though it is a widely known result that the amount of instruction-level parallelism present in a program is limited when considering only basic blocks, even more if instructions are never reordered. It is likely that implementing a more aggressive strategy would improve the emitted code quality significantly.

Automatic Computation of Data Size

While the NetVM model allows to fetch and store any data size (≤ 32 bits), registers are 32-bit words. This is a problem for the X11 processor that works natively on 16-bit words because of the larger overhead required to perform 32-bit operations, while often these can be correctly carried out using only 8 or 16 bits.

Although this is clearly a limitation of the NetVM model that does not explicitly support different data sizes, we decided to implement an heuristic algorithm in the X11 back-end that tries to assign to each NetVM register the optimal, minimum size while preserving the program semantics. In the long term, this issue points out the necessity of a revision of the NetVM model that will involve the addition of new NetIL opcodes to provide the NetVM with hints about the appropriate data size.

4.6. Conclusion

This Chapter presented the design and implementation of an optimizing multi-target compiler and run-time system for the NetVM model, in order to demonstrate its capability to enable the portability of packet processing applications, while ensuring an efficient mapping on a wide range of heterogeneous target platforms. In particular, the compiler allows the translation of NetIL programs to native code of three different architectures, exploiting the hardware features available on real network processors.

Even if the problem of partitioning applications across multiple symmetric execution cores (e.g. like those of the Cavium Octeon network processor) has not been taken into account, experimental results reported in Chapter 7 show that the generated code has performances often better than those obtained from hand-written programs compiled with state-of-the-art general-purpose compilers.

5. Assessing the programmability of the NetVM

5.1. Introduction

In order to assess the capability of NetVM to be an effective platform for the development of real-world applications, a clone of the popular Network Intrusion Detection Sensor[3] (NIDS) Snort [46] has been designed and implemented for the NetVM. The choice of this type of application is due to its requirements in terms of intensive packet-processing capabilities, dealing with all protocol layers and performing deep packet inspection. In addition, IDSs are suitable for hardware acceleration because

[3] A Network Intrusion Detection Sensor (NIDS), briefly IDS, is a network monitoring tool designed to detect unwanted attempts at accessing, manipulating and/or disabling computer systems on a network

of their extensive use of regular expressions and lookup tables, which are often assisted by specialized coprocessors on physical platforms.

In this Chapter, the architecture of the application is presented, showing that NetVM provides a programming model that is general enough for supporting the development of very complex packet processing applications that can be seamlessly ported onto extremely different platforms. Indeed, experimental results reported in Chapter 0 show that the Snort clone for the NetVM can be executed without any change on two heterogeneous target architectures (namely the Intel x86 and the Cavium Octeon), with performances that are comparable with those of the original application running natively.

5.2. Related Work

The implementation of a complete Snort-like intrusion detection sensor on a network processor was first explored by [47] that presents a compiler for generating C code from a set of intrusion signatures to be executed on an Intel IXP1200 NPU. The choice of generating C code was dictated by the need of exploiting the available development toolchain. However, this solution requires recompiling the software offline (where compilers are available), and then the updated code must be downloaded to the physical platform. This solution is efficient in case of “stable” software, but it prevents the possibility to have live updates for the software (e.g. updated security rules). Our solution is also based on a compiler for translating a rule-database into executable code, but the generated program is represented through an abstract assembly language that has to be further translated into the target binary code by the NetVM JIT compiler.

Since network intrusion detection heavily relies on deep packet inspection functionalities, such as string and regular expression matching, great effort has been directed towards solutions for optimizing and offloading such processor intensive tasks

through efficient algorithms and specialized hardware modules or coprocessors [48][49][50][51][52][53]. Another approach is using optimized algorithms targeted over the physical hardware platform; for example, [54] proposes a modified version of the Aho-Corasick [42] string-matching algorithm that can be executed in parallel on several microengines of the Intel IXP1200 network processor.

Differently from other research projects, the proposed approach aims at validating the entire application instead of speeding up specific functions such as only string and regex matching.

5.3. The Snort Intrusion Detection Sensor

Snort [46] is the implementation of a passive network IDS that is the de-facto reference in this class of applications; hence it seemed an obvious choice to design our own IDS by keeping compatibility with its rules and alerting formats. In this way our IDS would get immediate benefit from the huge database of already-existing attack signatures, which would also offer an excellent testing environment.

Snort is currently capable of performing real-time traffic analysis and packet logging on IP networks. Its capabilities include protocol analysis and content searching, which can be used to detect a variety of attacks and probes, such as buffer overflows, stealth port scans, CGI attacks, SMB probes, OS fingerprinting attempts and many other security threats.

Snort uses a database of rules to describe the known attacks. Each rule is written on a single line of ASCII text through a flexible description language and is divided into two logical sections: the header and the options. The rule header contains an action, a protocol, source and destination IP addresses and netmasks, and source and destination transport-protocol level ports. The rule options section contains a series of keywords,

which can be used to specify additional tests that should be performed on a packet, such as searching for a particular string or a regular expression in the payload, or checking if the "code" field of an ICMP packet matches a particular value. If all the tests specified in a rule are verified, then the corresponding action is undertaken (e.g. sending an alert an/ord logging the packet). For example, the following rule:

```
log tcp any any -> 10.1.1.0/24 80 (content: "GET"; msg: "HTTP
GET";)
```

logs every packet coming from any host and directed to port 80 of any machine of the 10.1.1.0/24 network containing the 'GET' string. Such packets will be logged with a message saying "HTTP GET".

The architecture of Snort is highly modular: it includes a Decoder module, which aims at locating protocol offsets and field values, a set of preprocessors that are used to normalize the packet when needed (e.g. an SSL decrypter in order to allow the following code to perform tests on the content, an IP defragmenter module, etc.), and the detection engine, which is the core of the application, where incoming packets are matched against the rule database in search of a possible security threat.

The detection engine will use several strategies for reducing the amount of checks that must be performed on the packets. For example, Figure 24 shows an optimization of the content matching module based on the TCP destination port contained in the rules. In this example, content matching tests are grouped according to the TCP destination port contained in the packet, i.e. if the tcp.dstport is equal to 80, only the first, second and fourth rules (hence keywords "POST", "HEAD" and "GET") need to be tested by the content module. In case this control matches, these rules are set as "potentially matching", and the processing continues with further steps that aim at checking all the field rules. However, it is evident how this strategy (which in fact is more elaborated

than in this example) can reduce the amount of checks that needs to be performed on every packet. Besides, using similar techniques, rules can also be grouped by other packet-specific properties, like the source and destination ports for TCP and UDP packets, creating even smaller subsets.

More details on the Snort IDS can be found in [46].

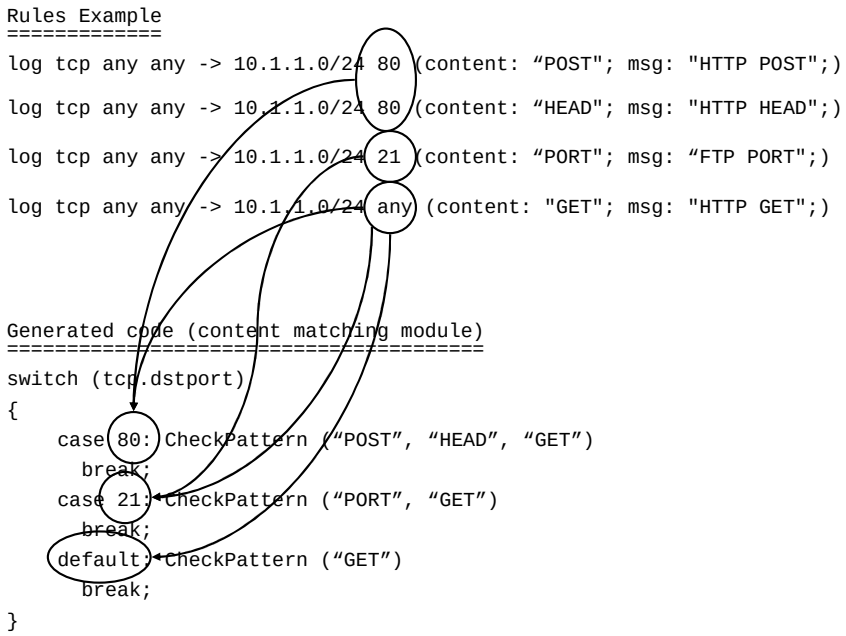


Figure 24. Rules optimization in Snort.

5.4. Architecture of the NetVM IDS Sensor

The IDS sensor for the NetVM is not a direct port of Snort; the two applications share almost no lines of code. Among the reasons for this choice are the lack of a C compiler for the NetVM and, more important, the belief that the C language is not always the best choice for highly packet-oriented processing applications. Our solution is based on a custom compiler that takes Snort rules and creates NetVM assembly. Even lthough the inputs and outputs of the application are the same as those of Snort (for instance, Table 2 shows the list of Snort keywords supported in our IDS sensor), its

internal architecture had to be redesigned from scratch in order to take full advantage of the NetVM paradigm, which tries to exploit the intrinsic modularization seen in packet-processing applications that are usually made up of several short and independent tasks. As the Snort rule format basically specifies tests that might involve the different protocols present in a packet, we decided to create different modules, instantiated on different NetPEs. Tests on each protocol are performed in the NetPE responsible for it, with the exception of some special functions (such as packet analysis and pattern matching) that are not associated to a single protocol and that are allocated to specific NetPEs. For instance, a rule such as "`log tcp any any -> 10.1.1.0/24 80`" will involve generation of code in different modules: the IP one will check that the destination address matches; the TCP module will be involved for checking the value of the TCP destination port, and so on. The rule will match only if all the tests are verified.

The final architecture is shown in Figure 25.

Table 2. Snort Keywords Supported by the NetVM IDS Sensor

Keyword	Description
msg	Message to use when logging
sid	Unique rule identifier used to keep track of developed rules
rev	Rule revision, used by Sourcefire
classtype	Type of attack the rule detects
reference	References to well-known application exploits the rule detects
itype	Search for a particular ICMP Type
icode	Search for a particular ICMP Code
icmp_id	Search for a particular ICMP ID
icmp_seq	Search for a particular ICMP Sequence number
dsize	Payload length
content	Search for a string in the packet payload
depth	Limit string search to a certain number of bytes
offset	Skip a certain number of bytes before string search
within	Limit string search to a certain number of bytes after a preceding string match
distance	Skip a certain number of bytes when searching after a preceding string match
nocase	Match a string case-insensitively
flow	Match a specific state/direction of a TCP connection
pcre	Search for a regular expression in the packet payload

The NetPE abstraction offers the possibility of an excellent modularization: each module is almost independent, and performance can be incremented by simply improving the code generation for NetPEs that represent the bottleneck, implementing

ad-hoc strategies to minimize the number of tests to be performed on a packet. For instance, some rarely used modules (e.g. ICMP counts for a few rules in the entire ruleset) use a very simple algorithm (linear search), while others implement smarter strategies. Global optimizations can also be implemented in the NetVM framework to be able to reduce the size of the target code.

This does not prevent global optimizations implemented in the NetVM framework to be able to reduce the size of the target code.

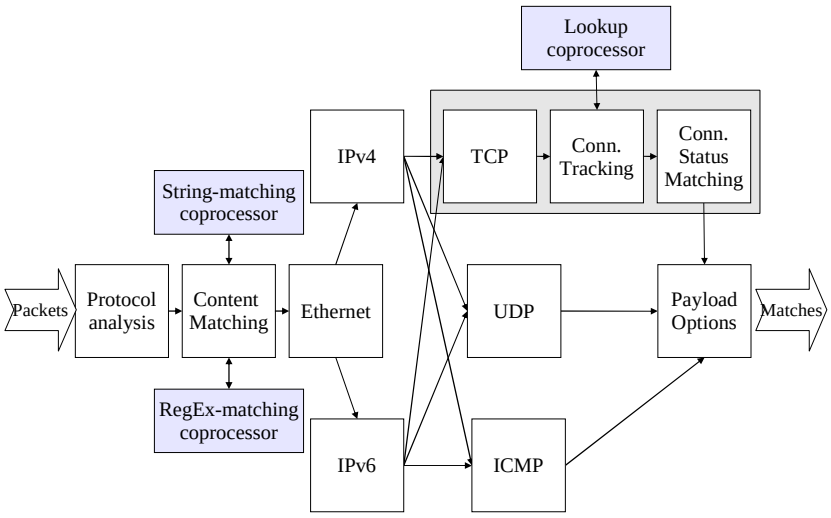


Figure 25. Architecture of the NetVM IDS sensor

As told in Chapter 2, NetPEs communicate among themselves through “exchange buffers”, i.e. meta-packets that, besides the packet buffer, contain additional data (e.g. time stamps) and a dedicated area called “info partition” where NetPEs can store state information that flows through the NetVM following the same path of the packet. Each module composing the IDS exploits the “info partition” for keeping the matching state of every rule and for communicating it to subsequent modules. In particular, as Figure 26 shows, the info partition is divided in two parts: the former contains a bit-vector, in which every bit represents a rule, while the latter is further organised into several 32-bit slots, each one containing data extracted from the packet, such as source IP address, port, etc.. When a packet enters the application, the bit vector is initialized to zero (i.e.,

no match) and the content matching module selects the group(s) of rules that are suitable for further processing by checking the proper patterns and by turning the corresponding bits to one. Then, the following modules should refine these controls by checking that all the conditions of each rule are verified. As soon as one condition does not match, the corresponding bit in the rules bitvector is reset; at the end, only the rules that have this bit set are matched.

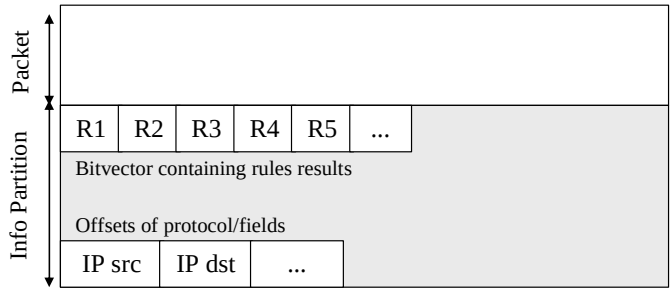


Figure 26. Exchange buffer: packet data and info partition

5.4.1. Packet-processing workflow

In our architecture, the processing of a new packet starts with the Protocol Analysis module that extracts information on the protocol headers that are contained into the packet and records the starting offset of the payload (if any). This piece of information is stored inside the “info partition” of the exchange buffer and is therefore made available to all the following modules in the chain. The next module is dedicated to Content Matching, which does some cross-layer checks in order to reduce the amount of strings to be tested on each packet and that matches the payload against a set of static patterns and regular expressions specified in the source rules. Since this task is the most processor-intensive, it relies on string and regular expression matching coprocessors provided by the NetVM architecture, which on general purpose platforms are emulated by software. The location of this module, almost in front of the processing chain, is due to performance reasons. In fact, the search is carried out by a modified version of the well-known Aho-Corasick algorithm [42] that allows several patterns to be searched at

the same time. As a result, if a pattern is found inside the payload, only the subset of rules based on it needs to be extensively verified.

Further modules will refine the processing by performing only the tests that are required on the subset of rules that have been selected as "possibly matching" in the previous modules. For instance, the IP, TCP and UDP modules group together all the rules that have the same addresses/ports, so that they only have to check each different combination of IP and netmask once. Another optimization consists in testing the destination address/port first, and then, if it matches, the source address/port. This approach is justified by the fact that, in real rule-sets, most rules have an unspecified (i.e.: "any", in Snort terms) source address and a precise IP as destination address, which stems from the fact that attacks come from anywhere, while the addresses of the servers in the internal network are well-known. Testing if the packet contains a precise destination address allows discarding a large number of packets immediately, reducing the ones that need to be further processed in order to detect a match.

The Ethernet module only checks if the packet contains IPv4 or IPv6, and sends it to the proper module, or just discards it in case the network-layer protocol is not supported. This module is extremely simple and does not provide any rule matching functionalities, since Snort rules do not support data-link layer tests (e.g., MAC-address based filtering).

The IPv4/IPv6 modules implement the tests over source and destination network addresses, while the TCP and UDP modules take care of checking the source and destination TCP/UDP ports of the packet, and the ICMP one checks all the possible ICMP options, which involve tests on the ICMP type, code, ID and sequence number.

The Connection Tracking and Connection Status Matching modules perform stateful TCP connection tracking, distinguishing who initiated the connection (i.e., server vs. client), the direction a packet is travelling in (i.e., from server to client or vice-versa)

and the state of the connection (i.e., established or still in the handshake phase). This task is performed with the aid of a lookup coprocessor that acts as an associative memory holding information on the current state of active TCP connections. Finally, the Payload module handles the matching of non-content payload-related options, such as tests on the payload size.

Connections among the various PEs are organized so that each incoming packet only traverses the subset of PEs dealing with the protocols it contains. This could be easily achieved through a scheme modelled after the TCP/IP protocol stack, as shown in Figure 4. This architecture has many advantages. First, each protocol is analysed only once. Second, the knowledge of a protocol is embedded in a single place, making the debugging easier and improving the handling of a protocol. Furthermore, the addition of a new protocol simply requires a new NetPE to be inserted in the chain (and the compiler to be updated to generate the new code for the NetPE). Third, the number of traversed NetPEs is small, i.e. packets traverse only NetPEs responsible of protocols that are present in the packet (i.e. an UDP packet will not traverse the NetPE dedicated to TCP), with a clear advantage from the performance viewpoint. Fourth, the architecture is suitable for pipelining. At the moment, the application handles one packet at a time, but potentially it could handle more packets if NetPEs can be instantiated on different physical execution units (e.g. in case of the Octeon multicore chip).

5.4.2. The code generation process

The traditional approach in intrusion detection applications is usually based on iterating over the rules that are represented in memory as complex data structures. For our IDS we decided to follow a different approach to the problem. In our implementation, rule checks are directly embedded in the code. In particular, instead of producing static programs that iterate over data structures in memory, the code directly

implements all the checks needed for matching packets against specific portions of the rules. Such a choice is based on the consideration that rules data remains constant throughout the execution of the program and such information can be exploited in order to emit checks (i.e. branch instructions) based on constant values (instead of checks based on values loaded from memory), producing more efficient code and opening the way to further optimizations. Since the resulting program is almost totally created at run-time, the entire code must be regenerated in case some rules change.

5.5. Conclusion

In this Chapter the implementation of a network intrusion detection sensor for the NetVM platform has been presented, in order to demonstrate that the NetVM programming model is suitable for creating complex packet-processing applications.

The current status of the IDS sensor is not as mature as the original Snort. For instance, some features (such as the IP defragmenter and TCP flow reassembly) are missing, and some application-layer keywords in the rule language are not supported. However, the objective was not to create a perfect clone of Snort, but to implement a reasonable proof-of-concept application for demonstrating the validity of the NetVM model. From this point of view, results are interesting, since NetVM primitives (i.e. the NetIL instruction set and the abstraction provided by virtual coprocessors) allow to effectively handle packet processing at all networking layers. Moreover, experimental results reported in Chapter 7 show that the runtime performances achieved are almost comparable with those of the native Snort.

On the other side, it is worth noting that NetIL is not a suitable language for programming the NetVM by hand, since it sits at a too low level of abstraction for a programmer (i.e. it is comparable to an assembler language), and its stack-based nature strongly limit its readability. However, this should not be considered a limitation of the

NetVM model, which is by design based on a mid-level programming language and aims at being an ideal target for several high-level programming languages.

6. Flexible Generation of Packet Filtering and Field Extraction Programs

6.1. Introduction

In order to demonstrate the possibility of decoupling the logic of a packet processing application from the knowledge of the actual format of the supported network protocols, while still ensuring runtime performances that are comparable with those of equivalent applications relying on hardcoded protocol descriptions, a compiler for the dynamic generation of packet filtering and field extraction programs has been designed and implemented.

Both filtering and field extraction rely on packet demultiplexing, i.e. a functionality for recognizing the full sequence of protocol headers contained in network packets. For example, a filter on “TCP” would first check whether the data-link frame contains an IP header, then it would check the IP header for a TCP header indication. Finally, if such sequence of conditions is completely satisfied, the corresponding action is triggered.

Similarly, extracting the values of the source and destination ports of the TCP header requires to first check if packets contain a TCP protocol header (i.e. a filter on TCP is applied) and then the actual values of the desired fields can be loaded from the packet buffer and made available to the user for further processing.

Demultiplexing programs implementing high level filtering predicates are usually generated by a compiler through routines, hardcoded in the compiler itself, that emit a sequence of checks on the values loaded at specific offsets of the packet buffer. For instance, such approach is taken by the libpcap library, which provides an API for the translation of simple filtering rules into a program for the BPF virtual machine [55]. The lack of flexibility in supporting new protocols, which requires the compiler to be extended (i.e., rewritten), represents a problem from the maintainability point of view. For example, in order to support a previously unsupported protocol, the compiler must be modified in several points: (i) new tokens representing the names of the new protocol and its fields must be added to the lexical scanner of the parser, (ii) the code generator routines must be extended for generating the proper checks on header fields, and (iii) already working routines must be made aware of the newly supported protocol.

The compiler presented here overcomes such limitations by decoupling the code generation process from the knowledge of the format of protocol headers, which resides in an external NetPDL database. In particular, NetPDL protocol descriptions are translated into packet demultiplexing programs that implement high level filtering rules expressed in the Network Packet Filtering Language (NetPFL). The generated code can be directly executed on any implementation of the NetVM virtual machine.

6.2. Generating Packet Filtering Programs from NetPDL and NetPFL

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 27. 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.

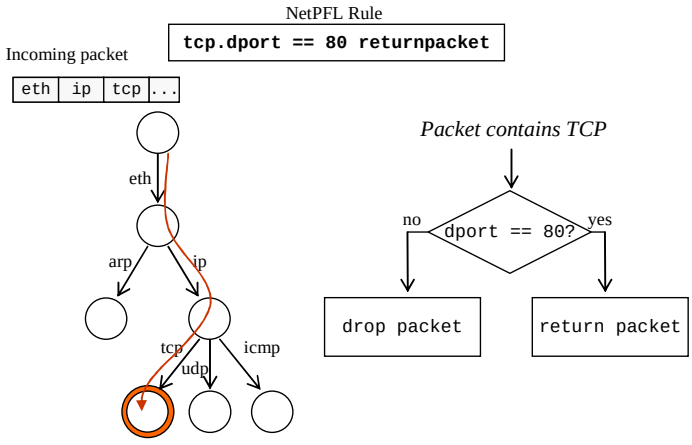


Figure 27. Filtering program as the composition of (a) a packet demultiplexing section and (b) a section for checking conditions on the target protocol fields

6.2.1. 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.

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 [56]. 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 28 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.

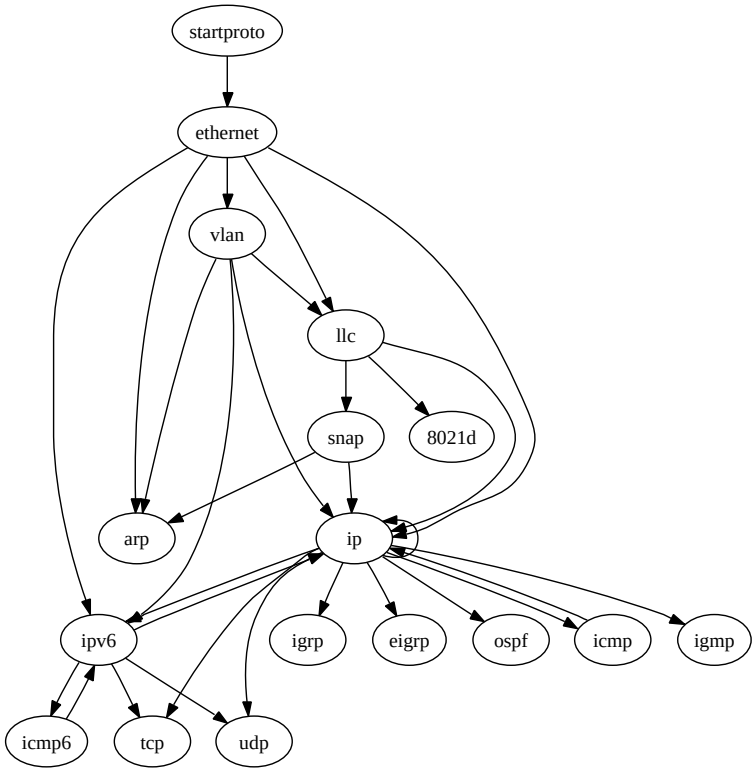


Figure 28. Protocol Encapsulation Graph.

6.2.2. Packet Demultiplexing

In the proposed 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 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 29. Code Generation Algorithm.

Given such considerations, our strategy for generating a packet filtering program through NetPDL is presented in the algorithm of Figure 29. 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 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 30 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).

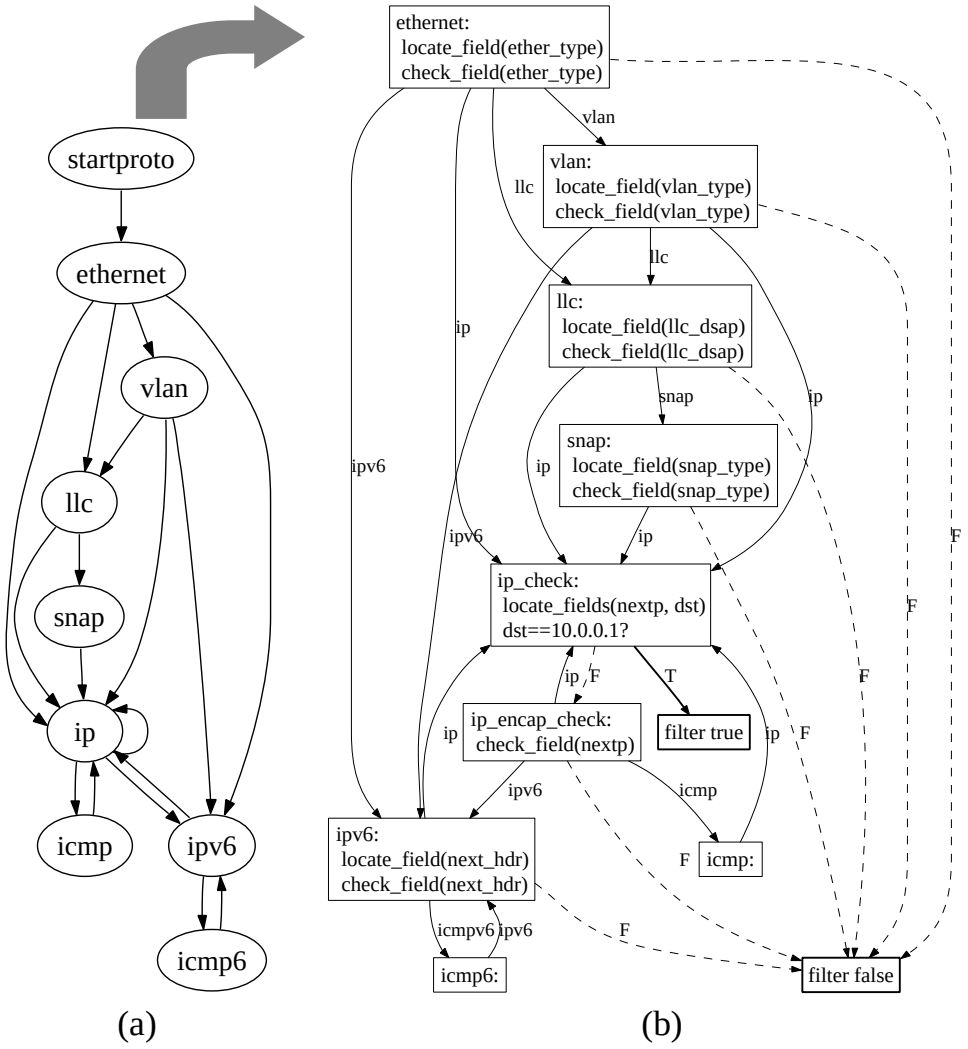


Figure 30. (a) Demultiplexing Paths and (b) Control Flow Graph for the filter "`ip.dst == 10.0.0.1 returnpacket`"

6.2.3. 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 (4).

$$Offs(Field_i) = Offs(Field_i\text{-}1) + Size(Field_i\text{-}1) \qquad (4)$$

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 (6), 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 6.3.3 will provide more details on such topic.

6.3. The Compilation Process

The techniques described in the previous section have been implemented 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.

6.3.1. 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 29 for each sub-expression referring to the same protocol, and by linking together all such portions of the program, as shown in Figure 31. The optimization of composed filters is left to future work.

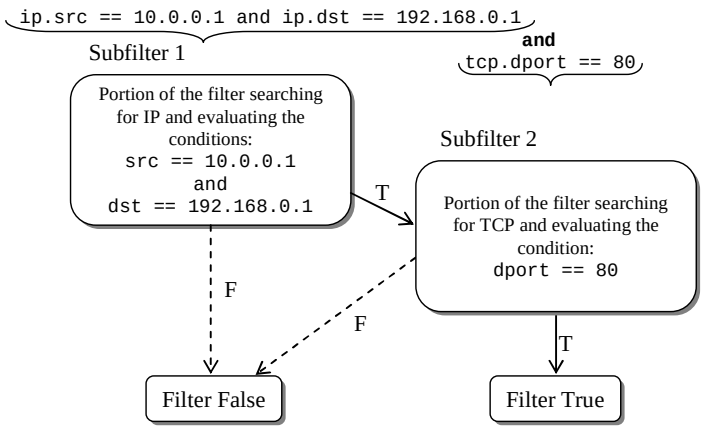


Figure 31. Composed filter.

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

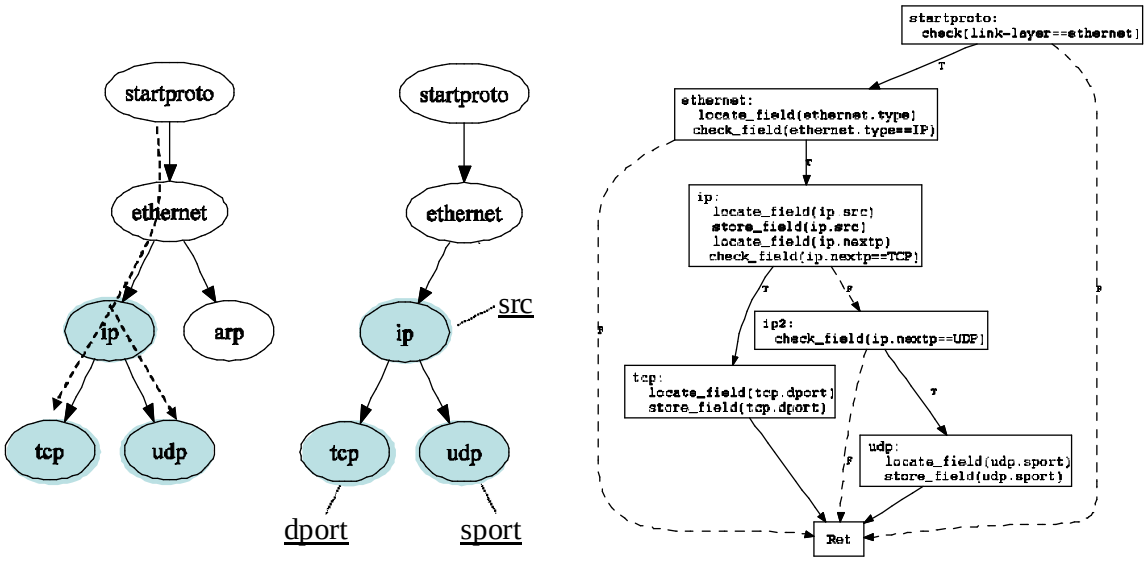
translated into masking operations on values loaded from the packet buffer. Finally, structured control flow constructs such as if-then-else, and loops are lowered to explicit branch operations.

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

6.3.2. Field Extraction

In order to handle field extraction rules, the code generation mechanism described so far is extended with the possibility to record the actual offset and size of the fields specified in a NetPFL `extractfields()` statement. In particular, since it is possible to request the extraction of fields belonging to different protocols, the algorithm described in Figure 29, is extended with the capability to visit in reverse post-order a more complex subgraph of the encapsulation graph, with more than a target protocol, because the generated program should be able to follow all the demultiplexing paths leading to each of them. Besides, for each target protocol, the appropriate statements are generated for storing the offset and size of the fields referenced in the NetPFL rule.

Such mechanism is exemplified in Figure 32, which shows the main phases involved in the generation of a program implementing the NetPFL rule "`extractfields(ip.src, tcp.dport, udp.sport)`". Figure 32A shows a minimal encapsulation graph containing only the Ethernet, ip, arp, tcp and udp protocols. Since the `extractfields()` rule specifies to extract some fields from the ip, tcp and udp headers (namely ip.src, tcp.dport, and upd.sport), these protocols are considered as the targets of the demultiplexing paths to be taken into account for generating the code. Such demultiplexing paths identify a subgraph of the encapsulation graph, which is shown in figure Figure 32B, with target nodes annotated with the names of the fields to be extracted.



(A) Encapsulation graph

(B) Demultiplexing subgraph

(C) Resulting Control Flow Graph

Figure 32. Code generation phases for the NetPFL rule "extractfields(ip.src, tcp.dport, udp.sport)"

Figure 32C shows the resulting control flow graph. The code is generated in a similar fashion respect to the case of packet filtering, by visiting the subgraph in reverse post-order (i.e. with a depth-first traversal where all the predecessors of a node are visited before the node itself), starting from target nodes: the `<format>` section of each protocol is translated into instructions for locating the fields required (i.e. those needed by encapsulation rules or requested for field extraction), while each `<encapsulation>` section is translated into branch instructions pointing to the next protocols. Besides, for protocols annotated with fields to be extracted, specific instructions for storing the offset and size of each field are generated. In particular, in order to communicate the extracted information to the user, the Info memory provided by NetVM is exploited, and the list of fields specified in an `extractfields()` rule is directly mapped on specific locations of the Info partition of the exchange buffer, as exemplified in Figure 33.

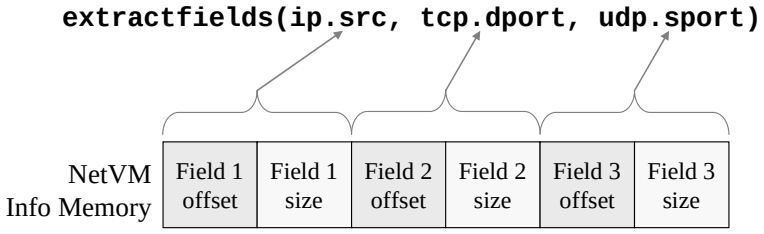


Figure 33. Allocation of fields on NetVM Info Memory locations

6.3.3. 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 [57] 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 the current work.

6.4. Conclusion

This Chapter presents the architecture of a compiler and a set of techniques for the dynamic generation of packet filtering and field extraction programs from NetPFL rules and NetPDL protocol descriptions, which constitutes the base for a novel approach to the development of packet processing applications whose logic is decoupled from the knowledge about the format of network protocols.

In order to minimize redundancies, the compiler deploys appropriate optimization techniques, leading to code that, in some cases, is completely equivalent to that of similar programs based on the hardcoded approach, as reported in Section 7.2. This demonstrates that the dynamic generation of efficient packet filtering and field extraction modules from NetPDL is feasible, with the advantage of adding support for new protocols or new encapsulation paths without changing the application code.

7. Experimental Results

7.1. NetVM Snort Evaluation

The capability of the NetVM snort front-end to generate NetIL code from a real Snort rule database has been assessed using an official ruleset provided by the Snort website in February 2007, which includes a total of 3058 rules, 1389 of them supported by the application. Such an apparent limitation is mainly due to the high number of rules requiring normalization and inspection of the URI field of HTTP headers (i.e. the "uricontent" option), which is a feature currently not supported. However, since the main goal was to demonstrate the ability of NetVM to allow the development of complex packet processing applications and not the complete compatibility with Snort features, such number can be considered a fair one, because it includes all the rules needing deep packet inspection functionalities (i.e. string and regular expression matching), and it is in line with the number of rules taken into consideration by other research works [51][52][53].

Table 3 shows the number of NetIL instructions generated from the abovementioned ruleset for each module of the IDS. From the table it is evident that the Content

Matching module is the one with the highest number of instructions. The reason depends on the complexity of the rules involving content matching options. In particular, when a match is found for the first content option, all other patterns eventually specified by the rule must be extensively searched inside the payload. Moreover, the first match could trigger more than one rule, making the generated code extremely complex.

Table 3. Number of NetIL instructions generated for each module

Module	Number of NetIL instructions
Analyzer	137
Content Matching	38872
ethernet	10
ip	4531
icmp	5547
udp	4806
tcp	5127
Connection Tracking	141
Conn. Status Matching	6228
Total	65399

The time needed by the rule compiler for generating the code is comparable with the one of the native Snort fed with the same database, containing only the rules supported by both tools. In particular, the NetVM based IDS compiles 1389 rules in 1,72 s, against the 1,25 s measured for Snort.

Since the runtime performances of the IDS depend on the capability of the NetVM framework to generate efficient code for the target architecture, detailed performance results will be reported in Section 7.3.

7.2. NetPDL/NetPFL Compiler Evaluation

This section assesses the ability of the NetPDL/NetPFL 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. 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 34.

```
  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 34. NetIL code generated for the filter `ip.src==10.0.0.1` with a minimal NetPDL DB

The corresponding BPF filter generated through the tcpdump tool is shown in Figure 35.

```
(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 35. BPF code for the filter ip.src == 10.0.0.1

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 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.

Since NetPDL supports a wide variety of protocols and cyclic encapsulations, the programs produced by the NetPDL/NetPFL 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 4. However, while BPF only identifies 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 will be shown in the next Section, 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 4. Number of Statements Generated by Different Compilers.

	Filter1	Filter2	Filter3	Filter4	Filter5
BPF	4	6	6	17	9
NetIL(reduced db)	10	14	23	76	26
NetIL (complete db)	292	491	487	1544	497

Currently, the NetPFL compiler is not optimized for speed in code generation. For instance, the libpcap compiler needs about 120μs 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 4 (first and third lines). It is worth recalling that the compilation time usually grows non-linearly with program size.

7.3. Performance Evaluation of the NetVM Framework

This section presents some tests that demonstrate the performance of the NetVM model and of its compiling infrastructure compared to other technologies. The tests are based on the two frontends available for the NetVM, that are the NetPDL/NetPFL compiler for packet filtering programs and the NetVMSnort Intrusion Detection Sensor.

7.3.1. Testing the x86 back-end

Tests on the x86 platform measure the performance of the code emitted by our compiler compared to two other targets. The first one is the code generated by the BPF virtual machine, which is able to generate native assembly through the WinPcap Just-In-Time compiler. Although the WinPcap JIT compiler is very simple compared to our compiling infrastructure, it provides a useful benchmark with a well-known and widely-used architecture. The second target is made up of a set of native programs created in C language and compiled with Microsoft Visual Studio, which represents the real touchstone of our solution. The native C filters use a custom macro to speed up byte-ordering operations, instead of using the standard `ntoh()` functions of the C standard library.

Five packet filters[4] with different complexity have been defined and their execution time has been profiled through the RDTSC assembly instruction available on the x86 architecture. Tests were performed on a Windows-based machine, equipped with a Pentium 4 processor, running at 3GHz with Hyper-Threading and 4GB of memory.

Results presented in Table 5 show that our compiler generates code that is faster than that produced by the other technologies under testing. Main reasons rely on the intrinsic properties of the NetVM model, which exports some useful information to the compiling infrastructure, thus enabling very effective, albeit simple, optimizations (such as compile-time constant swapping). Since the characteristics of packet-processing applications are taken into consideration in the entire compilation process, the NetVM compiler can perform more aggressive optimizations than its counterparts. Notably, this is obtained with a limited set of optimizations compared to commercial compilers (such as Microsoft Visual Studio). Additionally, results show that both the mid-level optimizations and those implemented in the x86 back-end introduce a substantial boost in performance (third column) compared to non-optimized code (second column).

Table 5. Filtering time on the x86 back-end (ticks)

Filter	NetVM no opt	NetVM opt	BPF	Native
1	23	7	36	8
2	26	12	39	26
3	30	15	39	13
4	52	39	76	61
5	35	21	43	34

[4] Filters, according to the well-known libpcap/WinPcap syntax are “ip” (filter1), “ip src 10.1.1.1” (filter2), “ip and tcp” (filter3), “ip src 10.1.1.1 and ip dst == 10.2.2.2 and tcp src port 20 and tcp dst port 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). The test packet was created so that filtering code was executed entirely before returning to the caller.

The capabilities of the x86 backend have been assessed also with the NetVM Snort IDS, fed with the same rule database described in Section 7.1, which includes a total of 1389 supported by the application. Table 6 shows the number of x86 instructions generated from the NetIL modules by the NetVM JIT compiler, and the actual size in memory of the target machine code.

Table 6. Number of x86 instructions and actual code size

Module	Number of x86 instructions	Code size (bytes)
Protocol Analyzer	163	613
Content Matching	268.667	1.130.250
Ethernet	20	104
IPv4	2.057	13.991
ICMP	2.906	16.737
UDP	1.838	13.173
TCP	2.100	14.442
Connection Tracking	261	1.271
Conn. Status Matching	2.097	14.054
Total	280.109	1.204.635

The performances of the IDS sensor have been assessed by measuring the time needed to process a trace of 10M packets captured on a real network and by comparing the results with those obtained running Snort under the same conditions (i.e., using the same rule database). All the tests were performed on a Dual Xeon running at 3,4 GHz equipped with Linux 2.6.20-15 SMP. The NetVM application was compiled Just in Time into x86 assembly, while Snort was compiled through GCC version 4.1.2. The

console output of the two tools was disabled in order to reduce every additional perturbation on the execution time that still includes the time needed for reading packets from file. Besides, all the features not supported by the NetVM Snort IDS (e.g. flow reassembly) were disabled in the native Snort. The tests have been repeated 12 times, and results have been averaged excluding the best and the worst run. Results are shown in Table 7.

Table 7. Throughput of the two applications

Application	Packets/Second
NetVM IDS (with x86 JIT)	70.344
Snort (native)	97.922

Results look interesting. Performances of the IDS sensor translated into native x86 code look promising, with the presented implementation running at 70% of the speed of the original Snort. Differences in speed are due to several factors: the IDS code that does not implements all the performance-oriented tricks of Snort because of the complexity of generating such this code in NetIL assembly. For instance, testing the destination port instead of the source port first makes a big difference in performance, and such these tricks are rather common in the original Snort. In other words, the performance penalties measured should be ascribed mainly to the algorithms used for analyzing packet data in the NetVM-based Snort, and refactoring the application would lead to performances comparable to those of the native Snort.

7.3.2. Testing the Octeon back-end

The first test on the Octeon back-end shows the results obtained with the same five filters already presented in the previous section. Due to the lack of a BPF JIT compiler for this platform, NetVM filters are compared only to handwritten ones, the latter using the GNU C compiler (GCC). Results (in clock ticks) are presented in Table 8. Also in this

case the code generated through the NetVM compiler is more efficient than that produced by the counterpart, thanks to the set of optimizations performed before emitting the code. In this case, the number of ticks is a good indication of the number of instructions emitted for each filter, because the Octeon processor is based on a MIPS pipelined architecture where most instructions are executed in exactly one clock cycle. These numbers can be further improved (although this is left to future work) by integrating a proper instruction reordering phase to avoid pipeline stalls.

Table 8. Filtering time on the Octeon back-end (ticks)

Filter	NetVM	Native
1	9	8
2	14	15
3	17	20
4	51	62
5	29	32

The NetVM Snort application has been profiled also on the Octeon platform. Although a direct comparison with the original Snort is not possible (processing algorithms are not exactly the same, and the original Snort does not run on the Octeon platform because of memory limitations), the main result is that NetVMSnort compiles and runs on the Octeon platform and is able to exploit native hardware coprocessors. This demonstrates the possibility of mapping even a complex NetVM application on this architecture, hence the validity of the NetVM model. Furthermore, Table 9 shows the comparison between the time spent in coprocessors (out of the total time used by the application to complete its job) between the x86 platform, where string-matching is executed in software, and the Octeon platform, where string-matching is executed through a hardware DFA engine, demonstrates that the NetVM model enables the efficient exploitation of native hardware features on platforms in which these are available.

Table 9. String matching performance on Octeon and x86

Platform	Percentage of the time spent in string matching
Octeon	3.79%
x86	13.44%

7.3.3. Testing the X11 back-end

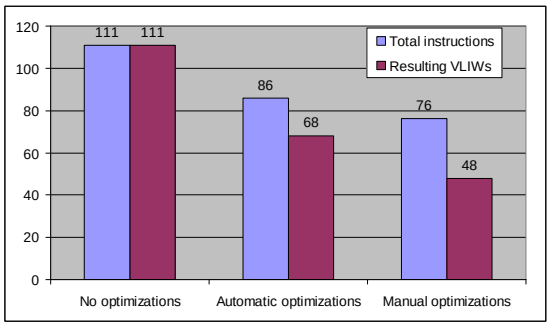
The X11 architecture presents many properties that make it predictable, allowing to exactly determine the behaviour of a program through off-line static analysis, without runtime benchmarking. The reason is that throughput is constant, as long as the code fits into the instruction memory of the systolic pipeline. Therefore, if the code is proven correct, a useful evaluation metric is the amount of instructions generated by the compiler. With a fixed size pipeline and a given number of passes, translating a program to fewer instructions allows more features to fit in the program with the same deterministic throughput.

For evaluating the X11 backend, two test programs were used: *(i)* a module of the NetVM Snort IDS, which performs L2-3-4 packet inspection and saves data for subsequent modules, and *(ii)* a simple packet filter that demultiplexes and counts TCP packets directed to port 80. Although these applications are small, we claim that the operations they perform are rather common in packet processing programs and stress several NetVM capabilities, using coprocessors and several kinds of memory.

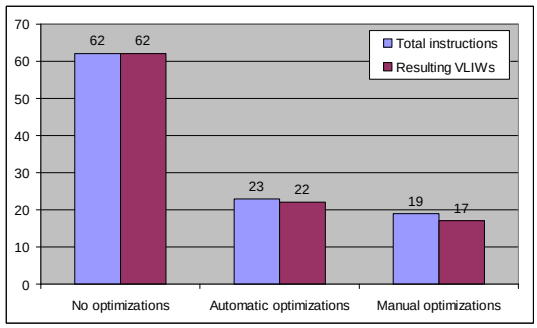
Since there are currently no other optimizing compilers for the X11, it is hard to the get the baseline results needed to evaluate the performance of the NetVM compiler. To get relevant results the source programs were first translated with all optimizations turned off. A second compilation was performed on the same source files, with all the automatic optimizations enabled. Afterwards the code, as already optimized by the compiler, was further processed by hand to apply a wider range of transformations, using standard optimization guidelines used by Xelerated. The same procedure was repeated keeping the

VLIW merging algorithm disabled in order to better appreciate its impact on the resulting code size.

Results are shown in Figure 36: the ones related to the IDS module are on the left (Figure 36a), while the ones related to the filter application are on the right (Figure 36b). Both the total number of instructions are shown, as well as the number of resulting VLIWs after instruction merging. As it can be seen, the number of instructions for the Snort application is 86/76 for the automated and hand-written cases respectively, while the corresponding numbers for the filter application is 23/19. After instruction merging, the results were 68/48 for the Snort module and 22/17 for the filtering.



(a) IDS Module



(b) TCP Filter

Figure 36. Code size for the test programs

Current results are encouraging: even with a prototype compiler and small applications, the instruction count obtained with the compiler is within 20% of the size of hand-optimized code before VLIW merging. Moreover, this was obtained by a proof-of-concept code that often used simple algorithms to speed up the implementation. We believe production-quality code can push this result even more. The differences between manual and automatic optimizations can be mainly ascribed to the simple VLIW merging algorithm employed, that does not perform instruction reordering, and to some missed copy folding opportunities. Both these issues can be addressed with standard techniques described in literature that do not require a redesign of the compiler framework to be implemented.

8. Conclusions

This work analyzes the possibility to introduce some degree of flexibility in the design and development of high-speed packet processing applications, like those that must be executed in network nodes subjected to elevated traffic rates and where runtime performances play a key factor.

The very general term “flexibility” has been considered in two specific contexts, i.e. (*i*) as the possibility to enhance the portability of packet processing programs for enabling the reuse of sofware solutions across heterogeneous processing architectures, while still ensuring the fulfillment of stringent performance requirements, and (*ii*) as the possibility to seamlessly integrate support to novel protocols and functionalities in packet processing applications, thus enabling the development of efficient and protocol-agnostic programs.

The former point has been addressed by refining the concept of Network Virtual Machine, i.e. a programming model based on an abstraction layer for the development of platform independent packet processing programs, which completely hides the characteristics of the hardware to the programmer, thus enabling source code portability

across a set of heterogenous architectures. A major contribution of this work relies on the demonstration of the fact that the use of a common abstraction layer, if well designed, instead of introducing a lack of runtime performances, enables the deployment of special purpose mapping techniques that concurrent and general purpose programming models do not allow, thus leading to programs that are both portable and efficient at the same time. This is possible by capturing in the programming model the characteristics of the peculiar application domain, allowing the programmer and a backend compiler to better share the knowledge on the actual semantics of the program, with the result of enabling the application of more aggressive optimization techniques.

On the other hand, the problem of decoupling the logic of packet processing applications from the knowledge of the format of network protocols in an efficient way has been addressed by leveraging the features of a language for the description of network protocols (NetPDL) and a language for the specification of packet filtering and field extraction programs (NetPFL). Using these components it is possible to create protocol-agnostic applications, however, in order to achieve good runtime performances, dynamic compilation techniques must be employed for the translation of the two languages into native code.

During this thesis the proposed technologies have been implemented and validated. In particular, a framework composed of a portable runtime environment and a compiler infrastructure, capable of JIT and AOT compilation, have been implemented. The framework allows to seamlessly port NetVM applications on three extremely heterogeneous architectures (i.e. the Intel x86, the Cavium Octeon and the Xelerated X11), with performances that are comparable and sometimes better than those obtained with alternative state-of-the-art compilers. This demonstrates the assumption that portability and efficiency can be achieved altogether, when domain-specific characteristics are adequately captured in the programming model.

The capability of the NetVM model to support the development of complex applications has been demonstrated by implementing a complete network intrusion detection sensor, which performs packet processing at all networking layers, leading to results almost comparable to those obtained by an equivalent application (Snort) running natively.

The possibility to efficiently decouple the logic of packet processing programs from the knowledge of the format of network protocols has been demonstrated by implementing a compiler for the NetPDL and NetPFL languages, which is capable of generating packet filtering programs to be executed by the NetVM runtime environment. Results show that in some cases the generated code is completely equivalent to the one generated by alternative solutions like the libpcap compiler for the BPF, which is based on hardcoded protocol descriptions. Moreover, thanks to the effectiveness of the NetVM compiler infrastructure, the runtime performances of packet filters generated from NetPDL/NetPFL can outperform those of equivalent hand-written programs compiled with state-of-the-art compilers.

Regarding the NetVM programming model, future work will be devoted to the investigation of the possibility to automatically partition packet processing applications on symmetric multi-core architectures, as well as to the analysis of problems related to the introduction of safety enforcing capabilities in the NetVM runtime environment.

The work related to the dynamic generation of packet processing programs from external protocol descriptions will be directed towards the analysis of techniques for minimizing the number of redundant checks in packet filters obtained by the boolean composition of basic filters (i.e. those based on conditions on fields of a single protocol), by leveraging the information provided by the presence of an “encapsulation graph” (see Section 6.2.1). Besides, constructs for correctly handling the presence of tunneling loops in packet headers are being included in NetPFL.

A. Network Processor Architectures

A.1. Xelerated X11

The X11 processor is a systolic processor. In medicine the term ’systole’ is used to refer to the rhythmical contraction of the heart, which sends blood throughout the whole body by pulsing. A parallelism can be drawn to computing systems where many processing units are linked together with hardwired interconnections and synchronized so that new data can be periodically sent into - and results can be extracted from the system, and a steady flow of data is sustained. Such architectures can be very regular and might be more easy to implement with VLSI technology.

A.1.1 The pipeline

The X11 is made of units called PISC (Packet Instruction Set Computers) which are connected to each other in a very long pipeline. Data enters the pipeline at the first PISC unit and exits the pipeline at the last PISC unit. Every cycle data is moved from a PISC

to the following one and every PISC performs an instruction on the data that has currently available, until the end of the pipeline is reached.

The PISC pipeline is augmented with Engine Access Points. These devices are interleaved between PISC blocks and serve as the access to external engines, which can be used to offload part of the computational complexity off the PISC pipeline. Figure 37 shows a pipeline segment.

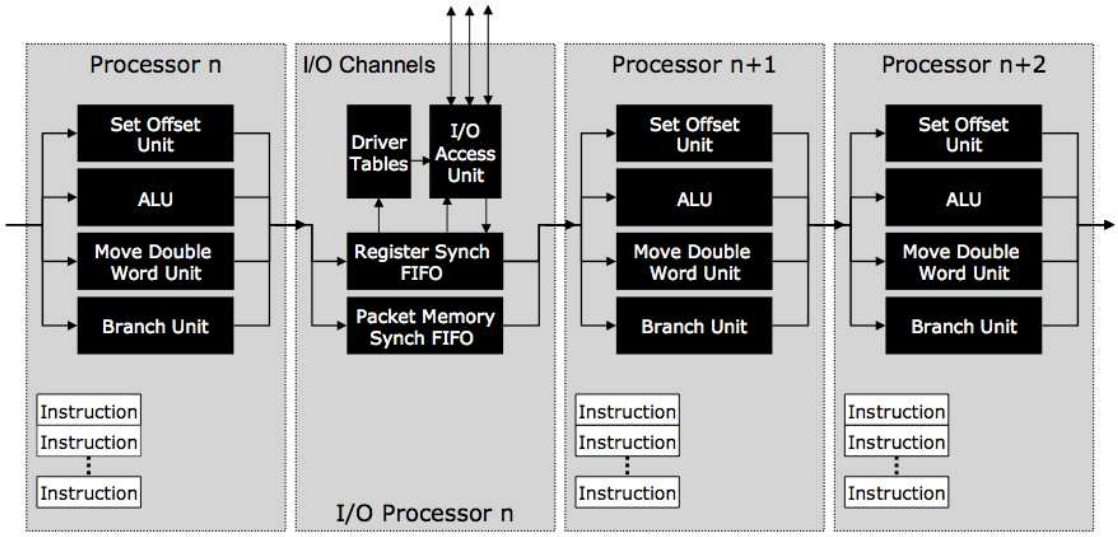


Figure 37. Detail of the X11 pipeline, showing 3 PISCs and an EAP. Courtesy of Xelerated AB. Excerpt from [58].

The entire pipeline is completely synchronous. There can be no stalls and no data can go lost in the PISCs and, under nominal operating conditions, in the EAPs. Packets enter the pipeline by the RX Arbiter, a device which feeds the first EAP (which, in turn, feeds the first PISC). Conversely, packets exit the pipeline by the TX Selector, which is fed by the last PISC in the pipeline. Since the whole machine is synchronous, the maximum rate of packets entering the pipeline is tightly related to the frequency of machine cycles, and (if no packets are dropped by the programmer) is equal to the rate of packets exiting the pipeline as well. There are a few consideration to make about the X11 processor that stem from the pipeline organization of the PISCs and the systolic architecture:

the amount of time required for processing every single packet is well-defined and known a-priori;

there is a strict instruction budget limit that affects programs written for the X11.

The first point derives from the fact that every packet follows exactly the same steps along the same physical units along the pipeline. No ’shortcuts’ can be taken to jump to a later stage: if the particular program execution for a given packets happens to use less instructions than the number of PISCs, we must wait for data to reach the end of the pipeline before the packet can be emitted. There can also be no waits of undetermined length in the pipeline, because every unit is able to complete its work within a single machine cycle. As a first approximation, no internal buffering is needed, or possible. The second point is a consequence of the finite length of the pipeline: once a packet has reached the end of it, processing is forced to terminate as there are no other execution units available. It is therefore impossible to execute a program that might require more instructions than the number of PISC processors in the pipeline. In order to let the programmer write longer, more complex programs than a single pass in the pipeline would allow, a loopback path is provided so that packets exiting the pipeline can reenter it for further processing. To preserve packet ordering, the number of pipeline passes is equal for every packet and is statically configured at compile time. If any quantity of the packets requires two or more pipeline passes for processing, every packet is bound to follow the same path and loop the same number of times. The maximum amount of iterations in the pipeline is fixed, and so it is the maximum possible execution time for any program. Using too many pipeline passes is undesirable. The number of loop interfaces is limited so they must be used sparingly and adding pipeline passes increases the processing latency. Finally, there is an upper limit on the number of pipeline passes given by the clock frequency of the X11 NP (which obviously cannot be scaled arbitrarily) and the packet rate requirement: if too many pipeline passes are required, the

packet insertion rate in the pipeline must be lowered. However the X11 NP is dimensioned so that multiple passes are possible while satisfying the wire speed requirements.

A.1.2 PISC units

PISC processors are the core of the X11 Network Processor. They are VLIW, 16- bit processors with a packet-oriented, RISC-like instruction set. They work on a general-purpose register file which holds operands and results. Data can be of 8-bit or 16-bit size. 32-bit operands are not directly supported. A special purpose register file holds the device registers, which are used to configure the pipeline and to hold other specific information. PISCs are made of 4 different functional units:

ALU, which handles arithmetical operation;

Copy unit, which can be used to move data between the packet memory and the register file, or different locations in the register file;

Jump unit, which is used to execute jumps (conditional and unconditional);

Load offset unit, which purpose is to load the available offset registers.

All the ALU operations must be performed over either 8 or 16 bits at a time. 32-bit operations can be implemented with multiple 16-bit instructions. On the contrary, the copy unit is able to move up to 32 bits at a time with a single instruction. The PISCs operate on very long instruction words (VLIWs) composed of four opcodes, one for every functional unit. At most one VLIW is executed in each PISC every machine cycle, as instructed by the RIP (Row Instruction Pointer) register in the device register file. There are no instructions that take multiple machine cycles to complete. In case a functional unit is not needed, its opcode in the VLIW instruction can be set to a no-operation. Every PISC has a private amount of code memory that holds multiple instructions. If we consider the whole PISC pipeline, the PISC instruction memory form

a rectangular matrix. Every PISC has associated a single column of memory, and the active row is specified by the RIP register. Linear code sequences are usually laid out along rows, so that consecutive PISCs execute one instruction of the sequence each. After the instruction for the current cycle is finished executing the data the PISC is working on is forwarded to the following stage of the pipeline and new data is received by the previous one, according to the systolic nature of the X11 NP. The systolic structure of the architecture makes it so that data must be forwarded in every machine cycle. This makes it impossible for a packet to "go back" to a previous PISC or to a processor that is not the immediate successor of the current one. Under this light it is important to understand what jumps mean on the X11 NP: the value of the RIP is modified so that the next PISC will execute an instruction that lies on a memory row different from the current one, but in no way the data flow between the pipeline elements can be altered. The code must be laid out in memory accordingly. The execution of any instruction in a specific PISC processor is inhibited if the Focus bit, held in one of the device registers, is set. In that case the PISC processor acts like a pass-through device for all the data it receives, forwarding them to the next stage with the correct timing. There are programmatic ways to set and reset the Focus bit. In the pipeline sequences of PISCs are interleaved with EAPs. Each group of consecutive PISCs between two EAPs (or between an EAP and the end of the pipeline) is called a PISC block.

A.2. Cavium Octeon CN38XX

The Octeon CNX3800 is a Network Service Processor (NP) targeted at network and network security applications. Like most NPs it integrates many processing units to exploit packet processing application parallelism. It can have from two to sixteen

processing cores *cnMIPS*, which are a simple, high-performance, dual-issue [5] implementation of the MIPS64®integer version 2 instruction set [59].

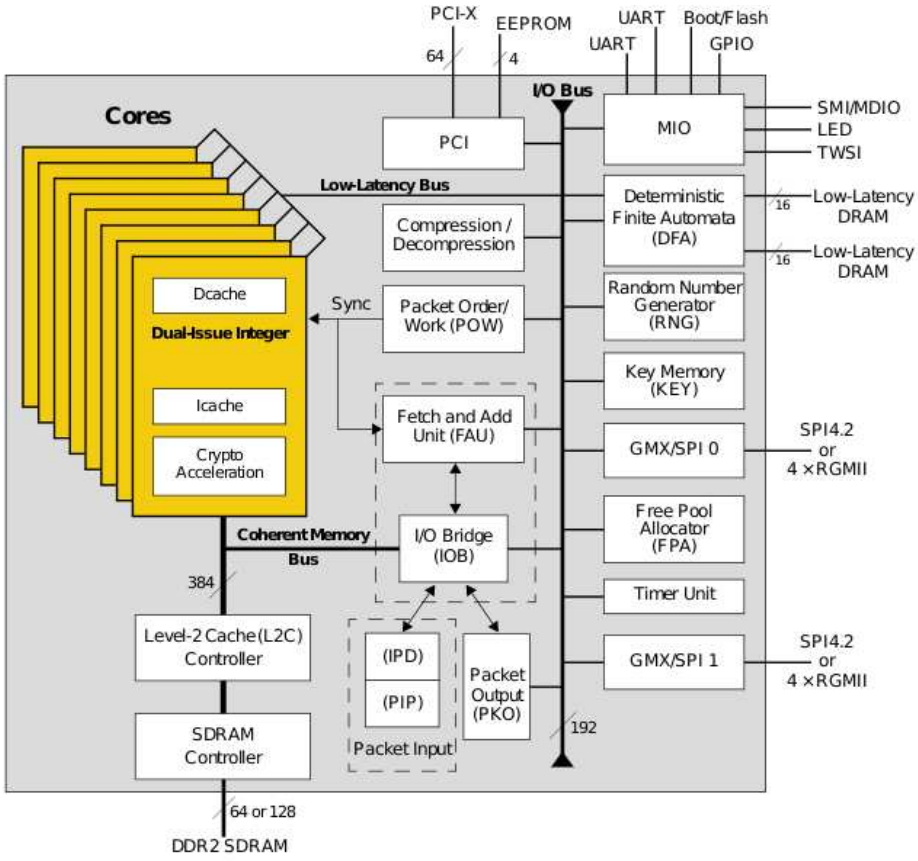


Figure 38. Architecture of the Cavium Octeon Network Processor

Figure 38 shows a block representation of the chip architecture. The left part (the cores, the coherent memory bus *CMB*, the level-2 cache and the DRAM controller) implements an on-chip multiprocessor and a coherent memory system. The right part contains the I/O bus and interfaces together with configurable I/O and processing

[5] A dual-issue processor is able to process and execute two instruction for each clock cycle

hardware units. This part helps the cores in handling packets arrival, queuing, dispatching and forwarding, besides of hardware implementation of many packet processing function (checksum, cryptography, etc.).

A.2.1 Overall workings

Before going through a deep analysis of the most important components of the chip, let's have a look to the path of a packet flowing into the system. In this way we will briefly introduce every component with his function and have a first understanding of the system internal workings and possibilities. There are many different algorithms for efficient searching. Packets arrive via any of the RGMII, SPI-4.2, PCI or PCI-x interfaces. The *Packet Input Processing Units (PIP)* has the task of storing packet data in on chip buffers or in DRAM together with information needed by software like the input port. This unit can also parse layer-two/IP packets for error condition and perform TCP/UDP checksum.

Upon arrival, packets are transformed in working units to dispatch to cores. For every packet a new work is created and queued in the *Packet Order Work unit (POW)*. The works' queue is the primary on-chip communication and synchronization mechanism. cnMIPS cores can become aware of works waiting for elaboration either with interrupt or polling and can request a work at any time. Both cores and hardware units can submit works to POW which then schedules them for the cores. So software receives packets by obtaining the associated work structures. As we will see in A.2.4 both hardware and software can tag works in several ways. Tags are used to implement synchronization and QoS mechanism.

There is a hardware unit, the *Free Pool Unit (FPA)*, which manages pools of pointers to available packet buffers. Hardware and software can allocate and free buffers independently. Queues used by cores to submit command to various on chip coprocessors are dynamically allocated memory chunks as well.

Cores receive work units and process them. When they finish their elaboration they either submit the work again to the POW and then to another core (in this way cores can be arranged in a pipeline fashion), or finally they can decide to send out the packet. Packet transmission is managed by another specialized unit, the *Packet Output Processing unit (PKO)*.

A.2.2 cnMIPS Cores

There are up to 16 cnMIPS Cores in OCTEON CN38XX. They are a dual-issue MIPS64® Version 2 integer instruction set implementation with also privileged instructions. Two instructions can be fetched, decoded and issued per. All the cores support a 5+ stage[6] pipeline (see figure Figure 39) with a clock rate up to 600 MHz. They also integrate a 32k 4-way instruction cache and a 8K 64-way data cache. They support conditional clocking for minimal power dissipation. There is a core with special and more privileged architecture (core 0) where the supervisor mode is implemented.

Besides the standard MIPS64® architecture some Cavium specific extensions are implemented, like several bit manipulation instructions, unaligned memory accesses, specific cryptographic instructions. Cores can be configured as either little or big endian. They do not support floating point arithmetic.

Each core has its own virtual memory space, which is completely private. We will see that there specialized mechanism which permit inter-core communication.

[6]We can identify five fuctional stage in the pipeline, but the last one uses more than one clock time

It is possible to run a complete operating system on cores, as well as industry standard C/C++ applications. Obviously core 0 with privileged instructions would be the supervisor core in an OS scenario. It is possible to partition cores at boot in a way that some can run a fully fledged operating system, while others can run a native networking application. Communication between the OS and the application could be achieved by means of specific OS drivers.

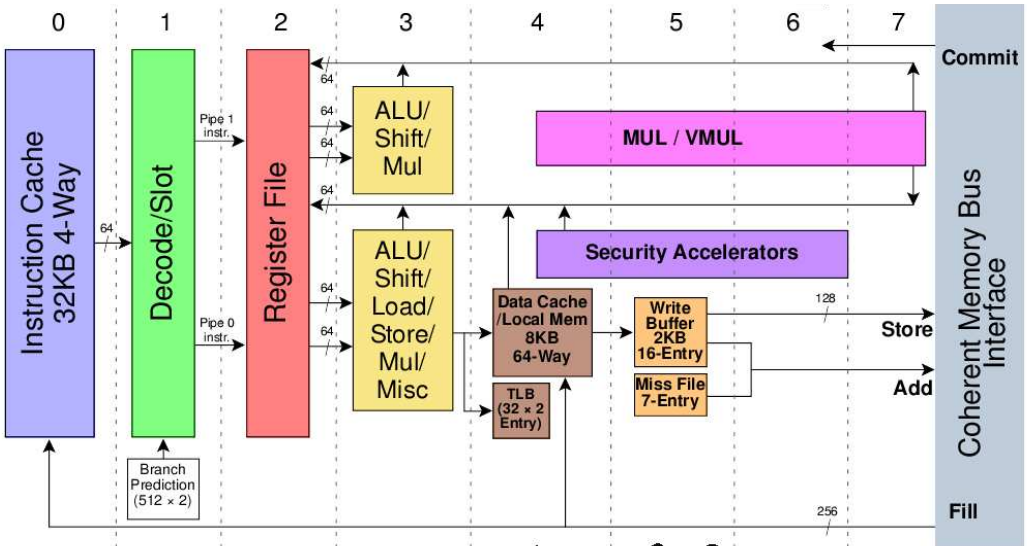


Figure 39. Octeon cnMIPS core pipeline

Cores use the Coherent Memory Bus to interface with memory and I/O. This bus also guarantees the coherence between the data caches of all cores. On the other hand, in order to communicate between them, the cores provide three main mechanism:

- Using works and POW work queuing units (explained in Section A.2.4)
- using shared memory regions (which requires software to handle locking)
- using interrupts to signal other cores when shared variables change

A.2.3 Packet Input Processing Unit

The Packet Input Processing Unit (PIP) receives packets from all the RGMII, SPI4.2, or PCI interfaces treating all the ports the same way. It can manage up to 36 input ports at the

same time. When a packet arrives, the PIP allocates and writes packet data into buffers in a format convenient to higher-layer software. The unit supports a programmable buffer size and can distribute packet data across different buffers to support large packet input sizes.

This unit also creates a work for the packet. The work contains a pointer to the buffered packet, packet error checks, and hardware parsing results (see A.2.4 for a deeper explanation of work structures). In fact the PIP unit can perform three kinds of automatic header parsing:

- **uninterpreted** parsing is skipped
- **skip-to-L2** parses various Ethernet-like L2 header and can determine whether IP is present in the packet
- **skip-to-IP** directly parse the contained IP packet

Normally the PIP unit writes packets in the L2/DRAM storing a pointer to the buffer in the work structure, but if the packet is smaller than 128bytes it is completely written into the work, hopefully in an on-chip buffer. The PIP units can be configured to write other useful values in the work structure:

- a *QoS* value, which gives packets different priorities and queues
- a *Group* value which decides to which cores the respective work will be scheduled. In fact cores can subscribe to different groups. The group value can be calculated from the input port or from the protocol, using hardware parsing data. This value can also be calculated from IP and TCP header field in order to give the same value to packets belonging to the same TCP-flow
- a *Tag* value, which can change the scheduling order of packets.

After writing all work fields PIP unit passes the work to the POW unit.

A.2.4 Packet Order Work Unit

The Packet Order Work Unit (POW) is a coprocessor providing important synchronization functions. A work is described by an associated work queue entry and may be created by hardware or core software. The OCTEON centralized packet input unit creates a work upon every packet arrival. The POW unit queues the work entries implementing eight input work queues. The POW can be configured to treat each queue in a different way, thus implementing different service levels.

Cores request work from POW. This unit selects the work for the core and return a pointer to the work-queue entry. All work is not equal, in fact the POW supports 16 different groups. Each piece of work is associated with a group. Each core has a configuration variable which select which groups can be submitted by the POW to that core. In this way it's possible assign different functions to different cores: for example packet processing may be pipelined from one group of cores to another group of cores performing to different stage of the processing. This is a very flexible and configurable system which enables programmers to better exploit the processor parallelism.

An other important field useful to order and synchronize related works is the tag field. There are three different tag types:

- **ORDERED** that guarantees ordering of works with this tag type
- **ATOMIC** that guarantees ordering and atomicity, so that two pieces of work with this tag type cannot be scheduled at the same time
- **NULL** that does not guarantee ordering

Core software can change the tag value via a tag switch transaction.

Typically a piece of work is scheduled to a core when core software executes a GET_WORK transaction to request a new work. The elaborations of works is sequential: no work can be scheduled to a core which is executing unscheduled work or is already elaborating some work.

A.2.5 Free Pool Unit

The Free Pool Unit (FPA) maintains eight pools of pointers to free L2/DRAM memory. The FPA hardware implements a data structure that approximates a logical LIFO for each free pointer pool. Both core software and hardware units use these pool. When a pool is too large to fit in-unit store the FPA creates a tree structure in DRAM using freed memory in the pool to store extra-pointer. Pool 0 is a special pool, since PIP stores packet data. Moreover PIP allocates work queue entries from a programmable pool. When one of these two pools becomes empty PIP cannot receives packets.

A.2.6 Packet Output Processing Unit

The Packet Output Processing Unit (PKO) gathers packets from L2/DRAM and sends them out on the RGMII, SPI4.2, or PCI interfaces. It can have up to 36 ports for sending packets to all destinations. The PKO unit supports up to 128 queues to buffer the packets to be sent out to the 36 available hardware ports. Each port can have a variable number of queues (up to 8) attached to it.

The system actually queues commands in these buffers instead of only packet data. In fact each packet transfer is a command. PKO performs a priority arbitration among the queues to decide which command is to be executed first.

PKO unit has also a specialized hardware to calculate the L4 checksum. In this case PKO can buffer the entire packet in its internal store. The PKO hardware only reads packet data from L2/DRAM once to send out a packet, unless it has to calculate the checksum for a TCP/UDP packet that is too large to fit in the internal buffering for a port (which can contain up to 1,5 KB). The unit can also recreate a complete packet from multiple segments stored in L2/DRAM, freeing the buffers of FPA containing the segments. All of these optional operations are specified in the command data submitted to the unit by the cores.

The PKO unit uses a priority algorithm that allows a configurable number of queues to be statically designated as high priority. When present, these priority queues must be the

lowest-index queues attached to the port. The lower the queue index, the higher the priority of the queue.

A.2.7 Deterministic Finite Automata Unit

The *Deterministic Finite Automata Unit (DFA)* is a coprocessor used to traverse graphs in memory. It can be exploited to implement fast hardware pattern matching algorithms.

A DFA is a state machine that receives as input a byte value (the DFA alphabet is made of the 256 possible values of 8-bit) which causes the transition from one state to the next. The states and the transition function can be represented by a graph, where each graph node is a state and different graph arcs represent state transitions for different input bytes. In the Octeon implementation each node in the graph is a simple array of 256 `Next Node Pointers`, one for each unique input byte value. Each `Next Node Pointer` contains a `Next Node ID`, which directly specifies the next node/state for the input byte, and a tag that can hold three values:

- **normal** nothing special for this node, continue traversing the graph
- **marked** the transition should be marked for later analysis by software. This is reported in the result.

terminal the next node is a terminal node and the graph traversal should stop

As shown in Figure 40, the DFA unit has three main components: the low-latency DRAM controller, 16 DFA thread engines (*DTEs*) and the instruction-input logic, which includes the instruction queue.

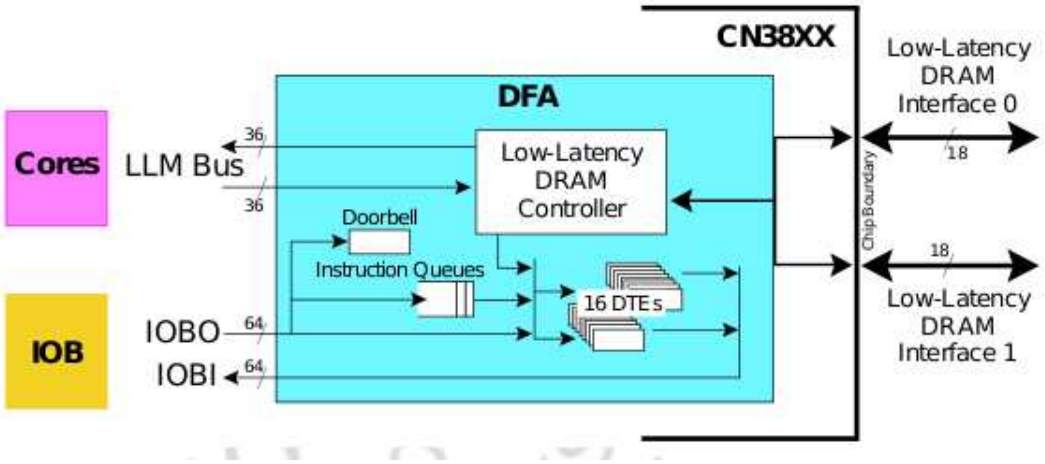


Figure 40. Architecture of the DFA Unit

The Octeon SDK contains specific tools which create the graph image to load in the low-latency memory (LLM) from the regular expressions. The user can load as many graphs as he likes, having the memory size as a limit. The LLM is is an external memory with an interface with a data rate equal to the core clock rate. The LLM DRAM controller can submit many operations to the memory at the same time and can also handle bank replication automatically in order to encrease data rates.

DTEs are independent coprocessors that can traverse graphs. When a core needs the DFA services, it must submit a command to the DFA instructions queue. The command contains the pointer to the graph we want to use, and a pointer to the data we are going to scan. When one of the DTEs is free, it fetches a pending command from the queue and starts walking the graph loading the packet data indenpendently from the DRAM. It also writes the scanning results back in DRAM as well.

Bibliography

[1] Ahmed A. Jerraya, "Long Term Trends for Embedded System Design," Digital Systems Design, Euromicro Symposium on, vol. 0, no. 0, pp. 20-26, Euromicro Symposium on Digital System Design (DSD'04), 2004.

[2] Intel. Internet Xchange Architecture network processors. http://www.intel.com

[3] Cavium Networks. Octeon network processors. http://www.caviumnetworks.com

[4] Xelerated. Xelerator X11 network processor. http://www.xelerated.com

[5] Bay Microsystems. Chesapeake network processor. http://www.baymicrosystems.com

[6] M. Baldi and F. Risso. Towards effective portability of packet handling applications across heterogeneous hardware platforms. In *IWAN 2005: Proceedings of the 7th Annual International Working Conference on Active and Programmable Networks*, Sophia Antipolis, France, November 2005.

[7] M. Baldi and F. Risso. A framework for rapid development and portable execution of packet-handling applications. In *ISSPIT 2005: Proceedings of the 5th IEEE International Symposium on Signal Processing and Information Technology*, Athens, Greece, December 2005.

[8] Lindholm, T. and Yellin, F. 1999 “Java Virtual Machine Specification. 2nd. Ed.” Addison-Wesley Longman Publishing Co., Inc.

[9] Miller, J. S. and Ragsdale, S. 2003 “The Common Language Infrastructure Annotated Standard.” Addison-Wesley Longman Publishing Co., Inc.

[10] R. Morris, E. Kohler, J. Jannotti and M. F. Kaashoek, “The Click modular router,” in *Proceedings of the 1999 Symposium on Operating Systems Principles*. December 1999.

[11] N. Shah, W. Plishker, K. Keutzer, “NP-Click: A Programming Model for the Intel IXP1200,” 2nd Workshop on Network Processors (NP-2), *9th International Symposium on High Performance Computer Architectures* (HPCA), Feb 2003.

[12] G. Memik and W. H. Mangione-Smith. NEPAL: A framework for efficiently structuring applications for network processors. In *Proc. of Network Processor Workshop in conjunction with Ninth International Symposium on High Performance Computer Architecture* (HPCA-9), Feb. 2003.

[13] Glen Myers, “Overview of IP Fabrics’ PPL Language and Virtual Machine,” White Paper, Online: http://www.ipfabrics.com/pdf/Overview_of_PPL_and_VM.pdf

[14] 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

[15] R. Ennals, R. Sharp, and A. Mycroft. Task Partitioning for Multi-core Network Processors. In *Compiler Construction*, volume 3443/2005 of *Lecture Notes in Computer Science*, pages 76–90. Springer Berlin/Heidelberg, March 2005.

[16] M. K. Chen, X. F. Li, R. Lian, J. H. Lin, L. Liu, T. Liu, and R. Ju. Shangri-la: achieving high performance from compiled network applications while enabling ease of programming. In *PLDI ’05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 224–236, New York, NY, USA, 2005. ACM.

[17] Turner, R. 2007. Understanding Programming Languages. Minds Mach. 17, 2 (Jul. 2007), 203-216

[18] Plezbert, M. P. and Cytron, R. K. 1997. Does “just in time” = “better late than never?”. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France, January 15 - 17, 1997). POPL '97

[19] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeugIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim – “LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation”

[20] Andreas Krall – “Efficient Java VM Just-in-Time Compilation”, in Proceedings of PACT’98, 12-18 october 1998

[21] Andreas Krall, Reinhard Grafl – “CACAO – A 64 bit Java VM Just-in-Time Compiler”, in Proceedings of the ACM PPoPP’97 Workshop on Java for Science and Engineering Computation.

[22] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, James M. Stichnoth – “Fast, Effective Code Generation in a Just-In-Time Java Compiler” In Proceedings of the ACM SIGPLAN ’98 Conference on Programming Language Design and Implementation, Vol. 33, No. 6, 1998

[23] Gupta, R., Pande, S., Psarris, K., and Sarkar, V. 1999. Compilation techniques for parallel systems. *Parallel Comput.* 25, 13-14 (Dec. 1999)

[24] Lee, E. A. 2006. The Problem with Threads. *Computer* 39, 5 (May. 2006), 33-42

[25] O. Morandi, F. Risso, M. Baldi, and A. Baldini. Enabling flexible packet filtering through dynamic code generation. In *ICC 2008: Proceedings of the IEEE International Conference on Communications*, Beijing, China, May 2008

[26] O. Morandi, F. Risso, G. G. Moscardi. An Intrusion Detection Sensor for the NetVM Virtual Processor. In *Proceedings of the The International Conference on Information Networking 2009* (ICOIN 2009) , Chiang Mai, Thailand, January 2009

[27] Johnson, E. J. and Kunze, A. 2002 Ixp-1200 Programming. Intel Press.

[28] F. Risso, M. Baldi, NetPDL: An Extensible XML-based Language for Packet Header Description, *Computer Networks* (COMNET), Vol. 50, No. 5, Elsevier, pp. 688-706, 2006.

[29] Computer Networks Group (NetGroup) at Politecnico di Torino, "The NetBee Library," August 2004. Available at http://www.nbee.org/.

[30] Computer Networks Group (NetGroup) at Politecnico di Torino, "Analyzer 3.0," March 2003. Available at http://analyzer.polito.it/.

[31] F. Risso, "NetPDL language specification," February 2007. Available at http//www.nbee.org/netpdl/.

[32] Computer Networks Group, "NetPFL language specification," August 2008. Available at http://www.nbee.org/doku.php?id=netpfl:index

[33] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM *Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[34] S. S. Muchnick. Advanced compiler design and implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[35] Z. Budimlic, K. D. Cooper, T. J. Harvey, K. Kennedy, T. S. Oberg, and S. W. Reeves. Fast copy coalescing and live-range identification. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 25–32, New York, NY, USA, 2002.

[36] C. W. Fraser, R. R. Henry, and T. A. Proebsting. Burg: fast optimal instruction selection and tree parsing. *SIGPLAN Not.*, 27(4):68–76, 1992.

[37] L. George and A. W. Appel. Iterated register coalescing. ACM *Trans. Program. Lang. Syst.*, 18(3):300–324, 1996.

[38] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. ACM *Trans. Program. Lang. Syst.*, 16(3):428–455, 1994.

[39] D. Bernstein, M. Golumbic, y. Mansour, R. Pinter, D. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compliers. In PLDI ’89: *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 258–263, New York, NY, USA, 1989. ACM

[40] Intel Corporation , 2008, Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture

[41] A. Korobeynikov. Improving switch lowering for the llvm compiler system. In *SYRCoSE 2007: Proceedings of the 2007 Spring Young Researchers Colloquium on Software Engineering*, Moscow, Russia, May 2007.

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

[43] J. Carlstrom and T. Boden, Synchronous dataflow architecture for network processors, *IEEE Micro*, vol. 24, no. 5, pp. 10–18, 2004.

[44] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[45] J. A. Fisher, “Trace scheduling: a technique for global microcode compaction,” *IEEE Transactions on Computers*, vol. 30, no. 7, pp. 478–490, July 1981.

[46] M Roesch, Snort - Lightweight Intrusion Detection for Networks, in *Proceedings of the 13th Systems Administration Conference* (LISA '99), Seattle, WA, November 1999, pages 229-238.

[47] Y. Charitakis, D. Pnevmatikatos, E. P. Markatos, and K. G. Anagnostakis, “Code generation for packet header intrusion analysis on the IXP1200 network processor,” in *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems* (SCOPES 2003), September 2003.

[48] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs", In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM01), April 2001.

[49] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. W. Lockwood, "Deep packet inspection using parallel Bloom filters," in *Hot Interconnects*, (Stanford, CA), pp. 44-51, August 2003.

[50] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. "Deterministic memory efficient string matching algorithms for intrusion detection". In *Proceedings of IEEE Infocom* 200, pages 333-340.

[51] S. Dharmapurikar, and J. Lockwood, "Fast and scalable pattern matching for content filtering", In *Proceedings of ANCS 2005*, Princeton, NJ, USA, October 26 - 28, 2005.

[52] 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 *Proceedings of the ANCS 2006*, San Jose, California, USA, December 03 - 05, 2006.

[53] Y. H. Cho, and W. H. Mangione-Smith, "A pattern matching coprocessor for network security", In *Proceedings of the 42nd Annual Conference on Design Automation* (DAC 05). San Diego, California, USA, June 2005.

[54] R. T. Liu, N. F. Huang, C. N. Kao; C. H. Chen, C. C. Chou, "A fast pattern-match engine for network processor-based network intrusion detection system", in *Proceedings of the International Conference on Information Technology: Coding and Computing* (ITCC 2004), Volume 1, pp. 97 – 101.

[55] A. Begel, "Applying General Compiler Optimizations to a Packet Filter Generator". 1996. Available online at http://www.microolap.com/downloads/pssdk/literature/begel96applying.pdf

[56] 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.

[57] 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.

[58] Gary Lidington. Programming a data flow processor. Available Online at http://www.xelerated.com/uploads/files/54.PDF, September 2003

[59] MIPS Technologies Inc., 2005. Mips64® architecture for programmers volume I: Introduction to the Mips32® architecture. Available online at http://www.mips.com/products/resource-library/product-materials/mips-architecture/