POLITECNICO DI TORINO
Repository ISTITUZIONALE

Towards Effective Portability of Packet Handling Applications Across Heterogeneous Hardware Platforms

Original
Towards Effective Portability of Packet Handling Applications Across Heterogeneous Hardware Platforms / Baldi, Mario; Risso, FULVIO GIOVANNI OTTAVIO. - 4388:(2005), pp. 28-37. (IFIP TC6 7th International Working Conference, IWAN 2005 Sophia Antipolis (FR) November 21-23, 2005) [10.1007/978-3-642-00972-3_3].

Availability:
This version is available at: 11583/1494645 since:

Publisher:
Springer

Published
DOI:10.1007/978-3-642-00972-3_3

Terms of use:

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

Publisher copyright

(Article begins on next page)

10 May 2026

Towards Effective Portability of Packet Handling Applications Across Heterogeneous Hardware Platforms

Mario Baldi, Fulvio Risso

Politecnico di Torino, Dipartimento di Automatica e Informatica, Torino, Italy
{mario.baldi, fulvio.risso}@polito.it

Abstract. This paper presents the *Network Virtual Machine* (*NetVM*), a virtual network processor optimized for implementation and execution of packet handling applications. As a Java Virtual Machine virtualizes a CPU, the NetVM virtualizes a network processor. The NetVM is expected to provide a unified layer for networking tasks (e.g., packet filtering, packet counting, string matching) performed by various network applications (firewalls, network monitors, intrusion detectors) so that they can be executed on any network device, ranging from high-end routers to small appliances. Moreover, the NetVM will provide efficient mapping of the elementary functionalities used to realize the above mentioned networking tasks onto specific hardware functional units (e.g., ASICs, FPGAs, and network processing elements) included in special purpose hardware systems possibly deployed to implement network devices.

1. Introduction

An increasing number of network applications performing some sort of packet processing are being deployed on current IP networks. Well known examples are firewalls, intrusion detection systems (IDS), network monitors, whose execution is must take place in a specific location within the network (e.g., backbone, network edge, on end systems) or, in some cases, be distributed across different devices. In general, such network applications must be deployed on very different (hardware and software) platforms, ranging from routers, to network appliances, personal computers, smartphones. In some cases, the whole range of potential target platforms is not even precisely and finally known at development time.

A development and execution platform for packet handling applications with features comparable to the ones of Java and CLR has been thus far not available. This paper reports on work aiming at designing, implementing, and assessing such a platform based on a *Network Virtual Machine* (*NetVM*), a new architecture for a (virtual) network processor in which execution of packet handling related functions is opti-

This work has been carried out within the framework of the QUASAR project, funded by the Italian Ministry of Education, University and Research (MIUR) as part of the PRIN 2004 Funding Program. Its presentation has been supported by the European Union under the E-Next Project FP6-506869.

mized. Specifically, when the NetVM is deployed on network processors or hardware architectures, packet handling related functions can be mapped directly on underlying special purpose hardware (such as ASICs, CAMs, etc) thanks to their virtualization in what are called NetVM coprocessors. This virtual device is programmed with an assembly language, or NetVM bytecode, that supports a set of interactions among the various blocks (e.g. memory, execution units, etc.) inside the NetVM. The project reported by this work addresses also the interaction between NetVM and external environment, e.g. how to download code to the NetVM, how to get the results of code execution, etc.

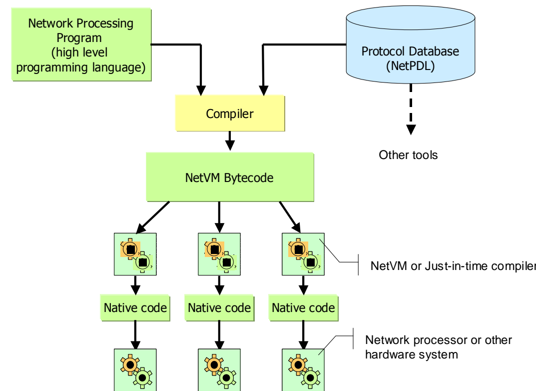

Fig. 1. NetVM framework.

Virtual machines are the basis for the "write once, run anywhere" paradigm, thus enabling the realization and deployment of portable applications. Even though from certain points of view the NetVM has a more limited scope than Java and CLR virtual machines (i.e., the NetVM targets a smaller range of applications), its goals are somewhat more ambitious. In fact, the latter aim at application portability across platforms that, while different from both hardware and software (i.e., operating system) point of view, are similar in being designed to support generic applications. Instead, the NetVM must combine portability and performance; this translates in the capability of effectively deploying available hardware resources (such as processing power, memory, functional units) notwithstanding the significantly different architecture and components of the various hardware platforms targeted.

The efficiency and portability of the NetVM has a significant by-product: it makes it a potential candidate for becoming a universal application development platform for network processing units (NPUs). Network processors combine high packet processing rates and programmability. However, programming NPUs is a complex task requiring detailed knowledge of their architecture. Moreover, due to the significant architectural differences, applications must be re-written for each NPU model. Deploying a virtual machine could help dealing with the diversity of network processors by offering a common platform for writing and executing portable applications. On the one hand, the NetVM hides the architectural details of the underlying NPU from the programmer. On the other hand, being designed specifically for network

packet processing, the NetVM has un-matched potential for effective execution on a hardware platform specifically designed for the same purpose.

NetVM programming is further simplified by the definition of a high-level programming language that operates according to packet descriptions realized with NetPDL (Network Packet Description Language) [3] and is compiled into native NetVM bytecode, as shown in the top part of Fig. 1. Once NetVM support be provided by commonly deployed network gear, distributed applications could be based on downloading NetVM code on various network nodes and possibly collecting the results deriving by its execution.

This paper is structured as follows. Alternatives for the implementation of the NetVM are presented in Section 2. Section 3 outlines the proposed NetVM architecture discussing its main components; performance issues are tackled in Section 4. Section 5 draws some conclusions and briefs current and future work.

2. NetVM Implementation

The NetVM aims at providing programmers with an architectural reference, so that they can concentrate on what to do on packets, rather than how to do that. This has been dealt with once for all during the NetVM implementation. This section focuses on how to implement the NetVM on both end-systems and network nodes.

Several choices are available, ranging from software emulation — NetVM bytecode is interpreted and for each instruction a piece of native code is executed to perform the corresponding function — optionally with specific hardware support (selected instructions can be mapped to specific hardware available on that platform), to recompilation techniques — e.g. an ahead-of-time (AOT) or just-in-time (JIT) compiler can translate NetVM bytecode into assembler specific for the given platform (es. x86, IXP2400, etc), therefore making use of the processor registers instead of operating on a stack.

A further option is to implement the NetVM architecture in hardware, i.e., the proposed architecture can be used as the basis for the design of a hardware device for network processing (e.g. VHDL can be used to create a new chip that implements the NetVM). Taking this option a step further, the NetVM code implementing a set of functionalities (e.g., a NetVM program that tracks the amount of IPv6 traffic) could be compiled in the hardware description of a (possibly integrated) hardware system that implements such functionality (e.g., an ASIC or an FPGA configuration). In other words, the NetVM could provide support to fast prototyping, specification, and implementation of network oriented hardware systems.

Since the NetVM design has been modeled after the modern network processor architecture, perhaps the most appropriate implementation option for the NetVM is an AOT/JIT compiler that maps NetVM assembler into a network processor's native code. This approach also solves one of the problems of network processors, which is their complexity from the programmability point of view.

3. NetVM Architecture and Components

The main architectural choices of the NetVM were driven by the goal of achieving *flexibility*, *simplicity*, and *efficiency* and built upon the experiences maturated in the field of Network Processing Unit (NPU) architectures since they are specifically targeted to network packet processing. The resulting NetVM architecture is modular and built around the concept of *Processing Element* (NetPE), which virtualizes (or, it could be said, is inspired to) the actual micro-engine of a NPU.

Processing Elements deal with only few tasks, but they have to perform them very fast: they have to process data at wire speed and in real time, they have to process variable size data (e.g. IP payload) or/and fragmented data (e.g. an IP payload fragmented over several ATM cells). In addition, they should execute specific tasks, such as binary searches in complex tree structures and CRC (Cyclic Redundancy Code) calculation with stringent time constraints.

Multithreading is an expected feature of a NPU, hence an objective of our architectural design: in fact packets are often independent from each other and suitable to be processed independently. For example, one of the first Network Processors — the Intel IXP1200 — is composed of six processing elements called Packet Engines. The larger the number of Processing Elements, the higher is the achievable degree of parallelism, since independent packets could be distributed to these units.

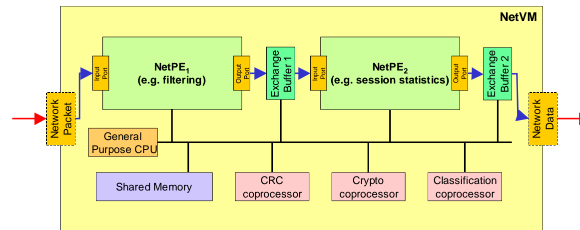

Fig. 2. NetVM configuration example.

A NetPE is a virtual CPU (with an instruction set and local memory) that executes an assembly program that performs a specific function and maintains private state. A NetVM application is executed by several NetPEs (for example, Fig. 2 shows an application deploying two NetPEs), each of which may implement a simple functionality; complex structures can be built by connecting different NetPEs together. Moreover NetPEs use specialized functional units (coprocessors, shown in Fig. 2) and various types of memories to exchange data. This modular view derives from the observation that many packet-handling applications can be decomposed in simple functional blocks that can be connected in complex structures. These structures can exploit parallelism or sequentiality to achieve higher throughput.

3.1. Processing Element (NetPE) Architecture

The general architecture of a NetPE includes six registers (Program Counter, Code Segment Length, Data Segment Length, Packet Buffer Length, Connection Table Length, Stack Pointer) in support to the processor operation, a stack used for instruction operands, a connection table whose purpose is outlined in Section 3.2, and a memory encompassing 4 independent segments (Section 3.3).

Like most existing virtual processors, the NetVM has a *stack-based design* where each NetPE has its own stack. A stack-based virtual processor does not encompass general-purpose registers as instructions that need to store or process a value make use of the stack. This grants portability, a plain and compact instruction set and a simple virtual machine. The consequence of this choice is that.

The execution model is *event-based*. This means that the execution of a NetPE is activated by external events, each one triggering a particular portion of code. Typical events are the arrival of a packet from an input, the request of a packet from an output or the expiration of a timer.

3.2. Internal and external connections

Connections are used to connect a NetPE with other NetPEs, with the physical network interfaces, and eventually with user applications. A NetPE can have a number of input and output *exchange ports* (or *ports* for the sake of brevity), each coupled to an exchange buffer. Each connection connects an output port of a NetPE to an input port of another one and is used to move data, usually packets, between the two.

Although the meaning of a connection is different, the connection model of the NetVM is similar to the one of Click[1]. Particularly, two types of connections are defined:

- *Push* connection: the upstream NetPE passes data to the NetPE at the other end of the connection. This is the way packets usually move from one processing function to the next one in network devices.
- *Pull* connection: the downstream NetPE initiates data transfer by requesting the NetPE at the other end of the connection to output a packet. Two options are provided for the downstream NetPE in case no packet is available: (*i*) it enters a wait state, (*ii*) an empty exchange buffer is obtained. For example, a NetPE that extracts packets from a buffer and sends them on an output interface uses a pull connection.

Also ports can be either push or pull. The NetVM runtime environment checks the validity of a NetPE interconnection configuration at creation time since there may be some illegal configuration, such as a connection between a push port and a pull port.

The number and type of ports of a NetPE is defined by the NetVM application and is maintained in the *Connection Table* within the NetPE, which is a read-only memory portion. The NetVM runtime environment fills out the connection table during configuration instantiation. Programs can use it to obtain, for every connection, the ID

[1] In Click [5] a connection is a direct call to a C++ method, while in NetVM it is a communication channel between two independent entities.

inside NetVM environment, the type (push / pull), and the direction (incoming or outgoing).

The NetVM communicates with external entities through of *NetVM sockets*. For example, if a NetVM is deployed inside the operating system of a desktop PC, external entities could be network devices, file streams or user applications that rely on the NetVM for low-level operations like filtering or network monitoring.

Applications that are intended to receive packets from a NetVM deploy a socket connected, through a push connection, to the push output port of a NetPE. The transfer of packets is initiated by the virtual machine (i.e., by the connected NetPE) and the application receives them through a libpcap-style [2] callback function. Alternatively, an application that is supposed to request data from a NetVM deploys a socket connected to the pull output port of a NetPE. Pull connections are appropriate to applications that retrieve tables, counters, flows, and other similar data.

An advantage of the socket/exchange port model is that transferred data is generic since exchange buffers are simple data containers; it follows that the application does not have any implicit information about the data that it receives, i.e., about data type, which must be provided in some other way.

3.3. Memory architecture

A NetPE has four types of memory: one shared among all NetPEs (*shared memory*), one for private data (*data memory*), one (local to the NetPE) that contains the program that is being executed (*code memory*) and one that contains the data (usually a network packet) that is being processed (*exchange buffer*). Shared memory can be used to store data that is needed concurrently by more than one NetPE (e.g., routing tables or state information). A NetPE is not compelled to use the shared memory: if it needs only local storage, only the Data Memory segment is used. This architecture allows to better isolate different kinds of memory and to increase efficiency through better parallelization of memory accesses. Memory addresses are 32-bit wide, although we do not expect to have such amount of memory (4GB) in network devices.

Since the NetVM may be potentially mapped on embedded systems and network processors, the use of high-level memory management systems like garbage collectors is not feasible. Therefore, the bytecode has a direct view of the memory. Furthermore, the memory is *statically* allocated during the initialization phase: the program itself, by means of appropriate opcodes, specifies the amount of memory it needs for being able to work properly. Obviously, these instructions can fail if not enough physical memory is present.

The flexibility lost with this approach is balanced by higher efficiency: the program can access the memory without intermediation thanks to ad-hoc *load* and *store* instructions. Specific instructions for buffer copies (a recurrent operation in network processing; some platform have even ad-hoc hardware units) are provided as well, either inside the same memory or between different ones. Moreover, knowing the position and the amount of memory before program execution allows very fast accesses when an AOT/JIT compiler is used because memory offsets can be pre-computed.

3.4. Exchange Buffers

Packets are stored in specific buffers, called *exchange buffers*, which are shared by two NetPE that are on the same processing path in order to minimize racing conditions (and avoid bottlenecks) when exchanging data. For instance, the NetPE1 in Fig. 2 will copy output data (e.g. the filtered packet) in the exchange buffer, which is then made accessible to NetPE2 for further elaboration (e.g. computing session statistics). Although, in principle, data can be moved from a NetPE to another through the shared memory, this could lead to very poor performance because this memory could become the bottleneck. Vice versa, exchange buffers provide a very efficient exchange mechanism between NetPEs that are on the same processing path.

In order to increase packet-handling efficiency, network-specific instructions (e.g. string search) and coprocessors may have direct access to exchange buffers. Instructions for data transfer (to, from and between exchange buffers) are provided as well. Furthermore, instead of moving packet data around, NetPEs can operate on the data contained in the exchange buffer, which are then "moved" from a NetPE to another. This is very efficient because exchanged buffers are not really moved; the NetVM guarantees exclusive access to them, so that only the NetPE that is currently involved in the processing can access to that data.

The typical size of exchange buffer is usually limited to some kilobytes; for larger data the shared memory can be used. This stems from the fact that this memory is often used to transport packets, although it can contain also generic data (e.g. fields, statistics or some generic state). In some cases, exchange buffers can contain also subportions of packets, as some network processors break packets into separate cells for internal transmission.

Usually, a NetPE has a single exchange buffer (i.e. it processes one packet at a time), although the NetPE specification does not prevent to have multiple exchange buffers. Exchange buffers are readable and writeable, although some particular virtual machine implementations could provide read only access for performance purposes or hardware limitations. Under these platforms an AOT/JIT compiler will refuse to build the NetPEs that perform write operations on packet memory.

3.5. Coprocessors

The NetVM instruction set is complemented by additional functionalities specifically targeted to network processing. Such functionalities are provided by *coprocessors* that, as shown in Fig. 2, are shared among the NetPEs. Making coprocessor functionalities explicitly available to the NetVM programmer is beneficial when the NetVM is executed on both general-purpose processors and network processors or special purpose hardware systems.

On general purpose systems coprocessors are realized by native code possibly implementing optimized algorithms. Code and data structures can be shared among different modules, thus granting efficient resource usage. For example, in a NetVM configuration with several NetPEs using the CRC32 functionality, the same coprocessor code can be used by all the NetPEs. If the implementation of the CRC32 coprocessor is improved, every NetPE benefits from it without any change in the NetVM imple-

mentation or in the application code. Also, more complex functionalities, such as string search or classification, can share data structures and tables among different modules for even better efficiency and resource usage. An example is the Aho-Corasick string-matching algorithm, which can build a single automaton to search for multiple strings as requested by different NetPEs.

On special purpose hardware systems, such as network processors, coprocessors can be mapped on functional units or ASICs, where present. Consequently, on the one hand the efficiency of NetVM programs is significantly increased when the target platform provides the proper hardware. On the other hand, writing NetVM programs represents a simple way of programming network processors or other special purpose hardware systems without having to know their hardware architectural details, yet while exploiting the benefits of their hardware specificities.

Communications with NetPEs is based on a well-defined, generic (i.e., not specific of a given processor) interface based on the IN and OUT assembly primitives, while parameters are pushed on the top of the stack. This guarantees a generic invocation method for any coprocessor without the need of any dedicated instructions; therefore coprocessors can be added without modifying the NetIL bytecode.

A "standard" coprocessor library (that includes a classification, a connection tracking, a string search and a checksum coprocessor, although some are still under development) is defined in the NetVM specification: a valid NetVM implementation should implement this library and each program using only coprocessors of the standard library should work on any valid NetVM. Additional coprocessors can be added to the library by NetVM implementations or third party libraries can be "linked" to a NetVM and used by applications that have been written to deploy the functionalities of non-standard coprocessors.

3.6. High Level Programming Language

NetVM programs are generally written in a high level programming language designed for networking applications, specifically for packet processing. One of such language (NetPFL) enables manipulations of packets and header fields whose format is described through the Network Packet Description Language (NetPDL) [3]. Although a detailed description of NetPDL and NetPFL is outside the scope of this paper, a sample is shown in Fig. 3 to offer a glance in the complexity of using the NetVM. The code instructs the NetVM to return on its exchange port number 1 all packets that, when parsed as Ethernet frames, contain the value `0x0800` in their `EtherType` field. In other words, this code implements a filter for IPv4 packets.

Fig. 3 shows both the syntax in the NetPFL language and the equivalent in the widely known tcpdump [2] packet filtering application. The comparison shows that, even though the NetVM provides the flexibility of a generic packet processing engine, programming a packet filter is not more complicated than specifying it for tcpdump, i.e., a utility specifically targeted and optimized for packet filtering. Hence, the increased flexibility of the NetVM is not traded for increased programming complexity, as well as for (significantly) lower performance, as discussed in the next section.

```
NetPFL:   ethernet.type == 0x800 ReturnPacket on port 1
tcpdump:  ether proto 0x800
```

Fig. 3. High-level code to filter IPv4 packets, in both NetPFL and tcpdump syntax.

4. Performance Evaluation

Although the current implementation of the NetVM is still in the early stages, a few numerical results are reported in this section in order to provide a first evaluation of the proposed architecture. To this purpose the NetVM is compared against the Berkeley Packet Filter (BPF) [1], probably the best-known virtual machine in network processing arena. Fig. 4 shows the assembly code required to implement the filter shown in Fig. 3, for both the NetVM and BPF virtual machines.

```
NetVM assembly

; Push Port Handler
segment .push

.locals 5
.maxstacksize 10
  pop           ; pop the "calling" port ID
  push 12       ; push the location of the ethertype
  upload.16     ; load the ethertype field
  push 2048     ; push 0x800 (=IP)
  jcmp.eq send  ; cmp the 2 topmost values; jump if true
  ret           ; otherwise do nothing and return

send:
  pkt.send out1  ; send the packet to port out1
  ret            ; return
Ends
```

```
BPF assembly

0) ldh  [12]             ; load the ethertype field
1) jeq  #0x800 jt 2 jf 3 ; jump to 2) if true, else 3)
2) ret  #1514            ; return the packet length
3) ret  #0               ; return false
```

Fig. 4. NetVM and BPF code to filter IPv4 packets.

A first comparison shows that the NetVM assembly is definitely richer than the BPF one, which gives an insight about the possibility of the NetVM assembly. However the resulting program is far less compact (the “core” is six instructions against tree in BPF). This shows one of the most important characteristics of the NetVM architecture: the stack-based virtual machine is less efficient of a competing register-based VM (such as the BPF is) because it cannot rely on a set of general-purpose registers. Hence, the raw performance obtained by NetVM cannot directly compete against the ones obtained by the BPF.

Table 1. NetVM Performance Evaluation.

Virtual Machine	Time for executing the “IPv4” filter (clock cycles)
NetVM	392
BPF	64

Table 1 shows the time needed to execute the programs reported in Fig. 4: as expected, the BPF outperforms the NetVM, mainly due to the additional instructions (related to the stack-based architecture) and the poor maturity of the code.

However, a NetVM is intended as a reference design and we do not expect its code to be executed as it is. In order to achieve better performance, NetVM code must be translated into native code (thorough a recompilation at execution-time, i.e., AOT/JIT compiling) according to the characteristics of the target platform. This justifies the choice of a stack-based machine, which is intrinsically slower, but its instructions are much simpler to be translated into native code. Performances are expected to be much better after a dynamic recompilation. The implementation of an AOT/JIT compiler is part of our future work on the NetVM.

5. Conclusions

This paper presents the architecture and preliminary performance evaluation of the NetVM, a virtual machine optimized for network programming. The paper discusses the motivations behind the definition of such architecture and the benefits stemming from its deployment on several hardware platforms. These include simplifying and speeding up the development of packet handling applications whose execution can be efficiently delegated to specialized components of customized hardware architectures. Moreover, the NetVM provides a unifying programming environment for various hardware architecture, thus offering portability of packet handling applications across different hardware and software platforms. Further, the proposed architecture can be deployed as a reference architecture for the implementation of hardware networking systems. Finally, the NetVM can be a novel tool for specification, fast prototyping, and implementation of hardware networking systems.

Some preliminary results on the performance of a simple NetVM program shows that other simpler virtual machines targeted to networking applications outperform the NetVM that, in turn, provides higher flexibility. Ongoing work on the implementation of a JIT compiler for NetVM code aims at reversing or at least reducing this performance discrepancy.

Since writing NetVM native code (bytecode) is not very handy, work is being done towards the definition of a high level programming language and the implementation of the corresponding compiler into NetVM bytecode.

Finally, in order to fully demonstrate the benefits, also in terms of performance, brought by the NetVM, further work includes the implementation of the virtual machine and its AOT/JIT compiler for a commercial network processor.

Bibliography

[1] S. McCanne, V. Jacobson, The BSD Packet Filter: A New Architecture for User-level Packet Capture. Proceedings of the 1993 Winter USENIX Technical Conference (San Diego, CA, Jan. 1993), USENIX.

[2] V. Jacobson, C. Leres and S. McCanne, libpcap, Lawrence Berkeley Laboratory, Berkeley, CA. Initial public release June 1994. Currently Available at http://www.tcpdump.org

[3] F. Risso, M. Baldi, NetPDL: An Extensible XML-based Language for Packet Header Description, To Appear in Computer Networks (COMNET), Elsevier.

[4] L. Degioanni, M. Baldi, D. Buffa, F. Risso, F. Stirano, G. Varenni, Network Virtual Machine (NetVM): A New Architecture for Efficient and Portable Network Applications, 8th IEEE International Conference on Telecommunications (ConTEL 2005), Zagreb (Croatia), June 2005.

[5] R. Morris, E. Kohler, J. Jannotti and M. F. Kaashoek: The Click modular router. Proceedings of the 1999 Symposium on Operating Systems Principles.