Network Virtual Machine (NetVM): A New Architecture for Efficient and Portable Packet Processing Applications

Loris Degioanni*, Mario Baldi*, Diego Buffa**, Fulvio Risso*, Federico Stirano***, Gianluca Varenni*
* Politecnico di Torino, Dipartimento di Automatica e Informatica, Torino, Italy
**Telecom Italia Labs - System On Chip, Torino, Italy
***Istituto Superiore Mario Boella, Torino, Italy

{mario.baldi,loris.degioanni,fulvio.risso,gianluca.varenni}@polito.it; stirano@ismb.it; diego.buffa@tilab.com

***Abstract*—A challenge facing network device designers, besides increasing the speed of network gear, is improving its programmability in order to simplify the implementation of new applications (see for example, active networks, content networking, etc). This paper presents our work on designing and implementing a virtual network processor, called NetVM, which has an instruction set optimized for packet processing applications, i.e., for handling network traffic. Similarly to a Java Virtual Machine that virtualizes a CPU, a NetVM virtualizes a network processor. The NetVM is expected to provide a compatibility layer for networking tasks (e.g., packet filtering, packet counting, string matching) performed by various packet processing applications (firewalls, network monitors, intrusion detectors) so that they can be executed on any network device, ranging from expensive routers to small appliances (e.g. smart phones). Moreover, the NetVM will provide efficient mapping of the elementary functionalities used to realize the above mentioned networking tasks upon specific hardware functional units (e.g., ASICs, FPGAs, and network processing elements) included in special purpose hardware systems possibly deployed to implement network devices.**

I. INTRODUCTION

Nowadays networks demand ever increasing packet processing speeds. Additionally, more and more intelligence is requested to the network, making the old approach, entirely based on defining custom ASIC for packet processing, no longer feasible. For instance, ASICs guarantee extremely high packet rates, but they cannot be reprogrammed and the time needed to develop (and prototype) a new chip is increasingly high.

A solution that can guarantee high packet processing rates, a new form of programmability and short developing time can be found in Network Processors. These are programmable chips whose architecture is particularly targeted to network packet processing. Their performances are obtained by a mixture of several components. First, a RISC-based processing core (which is well-known being very fast) that includes a reduced set of instructions, i.e. the ones that are most significant in packet processing (e.g. while floating point instructions are not present, there are special instructions for bit manipulation). Second, they implement a high degree of parallelism since there may be several execution engines inside the core that are able to execute multiple fragments of code in parallel. Third, there are some specialized hardware modules that are able to perform some of the complex tasks usually required in packet processing (e.g. table lookup engines).

As a general purpose CPU can be used to process packets, a general purpose VM (such as CLR or JAVA) can be used for this task too. However, not being optimized for this, they will never match the performance of a specialized VM, in terms of speed, memory requirements, and processing time. This paper provides a first report of a project aiming at the definition of the architecture of a virtual machine optimized for network programming, which is called Network Virtual Machine (NetVM). This work has the potential to:

- Simplify and speedup the development of optimized packet processing applications, such as traffic monitors, routers, firewalls;
- Enable efficient mapping of the execution of software modules performing specific tasks onto optimized components of custom hardware architectures;
- Provide a unifying programming environment for various hardware architecture;
- Offer portability of packet processing applications across different hardware and software platforms;
- Provide a reference architecture for the implementation of hardware (integrated) networking systems;
- Provide a new tool for specification, fast prototyping, and implementation of hardware (integrated) networking systems targeted to a specific packet processing application.

The work presented in this paper is based on previous work partly funded by Telecom Italia Labs. 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.

Section II, that outlines the motivations to and the potential benefits stemming from this work, further elaborated on the above implications. Section III discusses related work that broadly touches three areas: code portability across heterogeneous platforms, virtual machines, and, specifically, network and packet processing related virtual machines. The proposed NetVM architecture is outlined in Section IV and performance issues are discussed in Section V. Section VI draws some conclusions and briefs current and future work.

II. MOTIVATIONS AND BENEFITS

The NetVM aims to be a portable but efficient platform for demanding networking applications. While when the NetVM is executed by a general purpose CPU a NetVM program may be less efficient than a natively coded one, the situation may be reversed when the NetVM is executed by a network processor or a system with custom hardware and architecture. In fact, NetVM network-specific instructions can be efficiently mapped onto custom functional units (e.g., ASICs and FPGAs) that have been designed to optimally execute their tasks.

One of the problems of network processors is their complexity from the programmability point of view. Each device offers its own programming environment that usually includes a C-like compiler, which is different not only from vendor to vendor, but even between different lines of products of a single vendor. However, higher level approaches like C programming are not the ideal solution, because C has been invented for general purpose programming: it lacks many features that could help network development and presents some features that are not needed. For instance, the provided high-level languages lack of some of the standard functions (the ones that should not be related to network processing), while new ones are available (some that are needed to be able to exploit some hardware characteristics of the chip at best); therefore there is no guarantee about portability.

As a consequence, the most affordable way to program these devices (and to get efficient programs out of them) is to use native assembly language, which is time-consuming and error-prone. This requires not only a deep knowledge of the target machine, but also a non-negligible amount of time to program the device. Additionally, porting a program from a platform to another (even belonging to the same manufacturer) is a nightmare.

The idea behind the NetVM (Network Virtual Machine) architecture is to keep the high performance guaranteed by network processors while adding a reasonable degree of programmability. First, the NetVM defines the architecture for a new network processor, which is a reference architecture that wants to accomplish the most common tasks in packet processing. Additionally, it defines a way to extend this architecture by means of some additional functions (e.g. some hardware-based dedicated processors) in order to permit customization. Second, it defines the assembly language needed to program this virtual device and a set of specifications related to the interaction between every block (e.g. memory, execution units, etc.) inside the NetVM. Third, it defines how an application can interact with these components, e.g. how to download the code, how to get the results, and more.

The main goal of the NetVM is to provide programmers with an architectural reference, so that they can concentrate on what to do on packets, instead of how to do that. Each vendor can define a target compiler that translates this code into device-specific code. This guarantees a high degree of portability of the code between different devices, while keeping the performance characteristics of the network processors.

Taking this a step further, the NetVM architecture could be implemented in hardware, i.e., the architecture of the virtual machine could be used as the (basis for the) design of a hardware architecture for network processing (e.g., a network processor).

Analogously, the NetVM code implementing a set of functionalities could be compiled in the hardware description of a (possibly integrated) system that implements such functionalities (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.

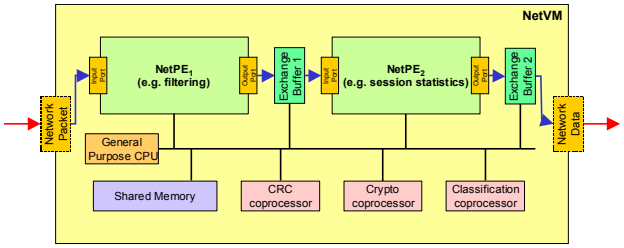

Figure 1. NetVM deployment example.

In Figure 1 there is an example of deployment of NetVM environment: the entire architecture is built around the processing element. As shown in the figure, each NetPE (similarly to a processing engine of a network processor) isolates a specific functionality and uses some specialized functional unit (coprocessor) and some shared memories to exchange data and packets among all virtual machine elements.

III. RELATED WORK

The most adopted approach to generate code portable among heterogeneous platforms is the one of retargetable compilers. Besides a bunch of alternatives [1][2][3], this is the approach adopted for Network Processor Units (NPUs) as well [4][5][6][7]. This methodology derives from the area of embedded systems, of which NPUs are considered evolutions.

A retargetable compiler is usually subdivided into two parts: a frontend and a backend. The frontend is in charge of source code analysis, generation of an intermediate representation, and machine-independent optimizations. The backend maps the machine-independent intermediate representation into machine-dependent assembly code. Such an approach is said to be retargetable since it is possible add one or more different target code generators to the backend.

From this point of view, a virtual machine can be considered an evolution of a retargetable compiler. In such a solution, the source language compiler is the frontend. It parses syntax, performs high-level optimizations and produces bytecode. The bytecode replaces the intermediate representation of a retargetable compiler, and the JIT compiler replaces the backend: it reads

intermediate language, validates it, performs machine dependent optimizations and generates native code.

Compared to a retargetable compiler, a virtual machine provides a better insulation among the modules: the fact that the backend is a completely separate entity ensures easier portability toward several targets, and the possibility for the developer of an hardware platform to support it without having to do with the compiler. Moreover, taking place just before code deployment (or even during code execution), the just in time compilation process is able to exploit a better knowledge of the target hardware and the application execution patterns, and as a consequence can implement more aggressive optimizations. Another advantage of using a virtual machine is that portability is provided at bytecode level: a library of networking functions can be used on all the supported platforms, without the need of recompiling or modifying the source code.

The best known virtual machines, like University of California P Machine [9], SUN JVM [8] or Microsoft CLR [10], are generic, system-independent platforms that run code generated by one or more languages. They are general purpose stack machines, with simple but standard instruction sets that provide instructions for load and store, control transfer, arithmetic operations, stack management and so on.

Although rich and thoroughly validated, these solutions are by their nature very generic, because they virtualize a general purpose CPU, therefore they often do not fit efficiently to application-specific tasks. Considering for example network processing, programs are organized in a way that is different from the one of desktop applications; furthermore, they make use of different primitives.

NetVM aims at providing specific support to network processing rather than to desktop applications: as seen in the previous paragraph, its architecture virtualises a NPU rather than a general-purpose processor. As a consequence, NetVM will never be better than Java or CLR in traditional tasks, however in the domain of network processing it provides advantages in terms of development time, portability and performance, both on traditional PC architectures and on programmable network devices.

Note that some virtual processors for network-related tasks already exist. The Berkeley Packet Filter (BPF) [11], described in section 2.2, is a well known filtering processor that supports packet capture; it is included in several Unix kernels, in the NPF driver [14] and in the pcap library [15]. SNAP [12] is a stack-based active networking language derived from PLAN, designed to be carried inside network packets and executed by the network nodes. SNAP was born to be interpreted, but it has been enriched with a JIT compiler for the IBM PowerNP network processor. Proposed by Kind et al. [13], this JIT compiler is the first attempt to apply just-in-time compilation to a NPU, and has the merit to demonstrate that this road is practicable with good results, especially in terms of performance. However, all these solutions are too limited and too tied to specific domains to be compared with NetVM. Both of them rely on very simple (and lean) virtual processor, which export minimal instruction sets and limited hardware abstractions. None of them, for example, allows backward jumps.

Another approach consists in subdividing a program into small pieces, each of which performs a single independent functionality; these pieces cooperate to create the overall application. Modularity in network processing applications has been investigated by a number of previous works, and is widely accepted nowadays.

[16] provides results that demonstrate the advantages of structuring applications for network processors in a modular way, and describes a system, called NEPAL, able to extract modules from an existing sequential network processing program. The simulations of this system with a series of applications, shows that modularization is an effective method to improve performance of network processing. Modularization, on the other hand, allows utilizing efficiently not only the strongly parallel NPU architectures, but also the widespread SMP desktop workstations.

Many existing systems make use of modularity; among them we can cite NetFilter [17], VERA [18] and Click [19]. Particularly, Click is a well-known software router, which introduced most of the concepts behind modular network processing, and which is used as a base and as a term of comparison by several following tools. NetVM too draws some concepts from its architecture. A Click router is made up of 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. It communicates with the external world sending or receiving packets by means of two kinds of connections: push or pull. In a push connection packet transfer is initiated by the source endpoint, while in a pull connection packet transfer is initiated by the destination endpoint. Variants of click have been proposed to address the issues of parallelism [20] and portability to network processors [21], and demonstrated the goodness and the feasibility of the modular approach under different hardware configurations.

VERA, another modular router, is a step forward for some of these issues: it has an architecture that includes a hardware abstraction, and it considers the issue of portability toward different processors (including NPUs). However, the overall architecture is strictly oriented to routing (starting from the hardware abstraction, that includes for example the switching element), with very few concessions to other kind of processing. Moreover, actual modularity is remarkably less pronounced than in Click. All in all, VERA privileges architectural hierarchy and pure performance rather than modularity portability.

IV. NETVM ARCHITECTURE

The main goals of the NetVM are **flexibility**, **simplicity** and **efficiency**. These objectives and the experiences maturated in the field of Network Processor architecture, determine the most important architectural choices of the project.

Packet processing, as well known, is suitable for a multi-stage pipeline or even for an array of processors: in fact processing can be easily divided across several processing units of moderate speed. Consequently, the NetVM has a modular architecture built around the

concept of *Processing Element* (NetPE), which virtualizes (or, it could be said, is inspired to) the actual micro-engine of a Network Processor.

Processing Elements (both hardware and software ones) have to deal with only few tasks, but they have to perform them in a challenging way: 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. the IP payload fragmented over several ATM cells). Consequently, a Processing Element must have advanced memory management and performing scheduling algorithms for the units that access memory directly. In addition, it 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 Network Processor, hence an objective of our architectural design: in fact packets are quite 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.

In our NetVM's architecture, a NetPE is a virtual CPU (with a proper set of instructions, and a local memory) that executes an assembly program that performs an individual function inside the NetVM and maintains private state. A NetVM application is assembled from several NetPEs (for example, Figure 1 shows an application made of two NetPEs), each of which implements a single functionality; complex structures can be built by connecting different NetPEs together. This modular view derives from the observation that many packet processing applications can be decomposed in simple blocks that can be connected in complex structures. These structures can exploit parallelism or sequentiality to achieve better performance. The modular approach is not new: other software solutions, like *Netfilter* [17] or *Click* [19] have demonstrated its goodness, and the parallel architecture (based on many simple microengines) of many network processors follows the same direction.

Figure 2 shows how NetPEs can be connected to build more complex structures, either pipelined or parallel. In the first case the packets emitted by the output interface of an instance are received by the input interface of the following one. In the second case the packets coming from a single source are processed in parallel by two or more NetPEs.

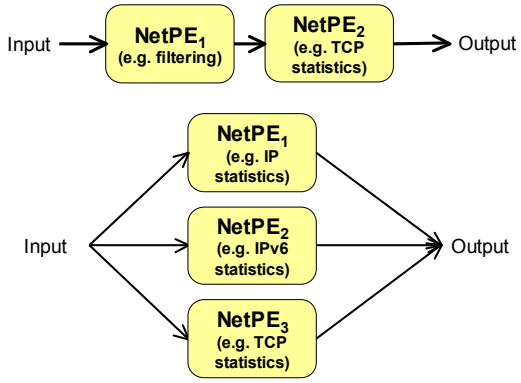

Figure 2. Multiple NetPEs organized either in a pipelined structure or running in parallel.

A. NetPE: the main Processing Element

The NetPE is the core of the Network Virtual Machine. It runs the NetVM bytecode, performing actual processing of network data.

Like most available virtual processors, NetVM, has a *stack-based design*. A stack-based virtual processor grants portability, a plain and compact instruction set and a simple virtual machine. The consequence of this choice is that no general-purpose registers are provided and all the instructions that need to store or process a value make use of the stack. Any NetPE has its own private stack.

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 endpoint, the request of a packet on an output connection endpoint or the expiration of a timer.

1) The NetPE Assembly

The NetVM instruction set derives from the one of a generic stack machine, with specific additions to support the particular architecture and application area of this virtual machine. Opcodes can be subdivided into several groups; the most important ones are listed in TABLE I.

TABLE I
EXAMPLE OF SOME INSTRUCTIONS PRESENT IN NETPE ASSEMBLY.

Category	Description	Example	Description
Initialization	Used to initialize the execution of a NetPE program	`set.share`	Set the size of the shared memory
Data transfer	Transfer data within memory	`dpcopy`	Copy a memory buffer from a portion of the memory to another
Pattern matching	Used to compare a value in a memory buffer against the top of the stack	`field.eq.8`	Compare the top of the stack with an 8bit field in memory
Flow Control	Used to control the execution flow	`jump`	Unconditional branch
Stack management	Used to manage the stack	`swap`	Swaps the two top elements of the stack
Aritmethig and Logic	Used to compute simple expressions	`ror`	Rotale right the value at the top of the stack

Being targeted to network processing, the NetVM instruction set includes a group of more specific opcodes, not present in traditional stack machines. Most of these opcodes reflect the instructions that some network processors provide to speed up packet header processing. For instance, TABLE II shows some of them.

TABLE II
EXAMPLE OF NETWORK-SPECIFIC OPCODES.

Opcode	Description
`find.bit`	Find bit set. Returns the position of the first bit inside the top of the stack.
`mfind.bit`	Find masked bit set. Return position of the first set bit in the top value of the stack, with mask.
`clz`	Count leading zeroes. Counts the number of consecutive zeroes from the MSB of the top of the stack.

Opcode	Description
`set.bit`	Set one of the bits of the top of the stack.
`clear.bit`	Clear or flip one of the bits of the top of the stack.
`flip.bit`	Flip one of the bits of the top of the stack.
`test.bit`	Conditional test on a bit of the top of the stack.
`field.c.t`	Compare a value of width *t* in the packet memory with the top of the stack. Branch if the condition *c* is satisfied. *t* can be 8, 16 or 32; *c* can be eq, ne, lt le, gt, ge. This instruction can be used as a support for protocol header parsing.

Since the NetVM may be potentially mapped on embedded systems and network processors, the use of high-level memory management systems like garbage collectors does not sound feasible. Therefore, the bytecode has a low-level, 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 the gained 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 a JIT compiler is used because memory offsets can be pre-computed.

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 when exchanging data. In order to optimise packet handling as fast as possible, network-specific instructions (e.g. string search) and coprocessors may have direct access to exchange buffers. Instructions for data transfer (to and from, or internal to exchange buffers) are provided as well.

B. Coprocessors

Like hardware network processors, that add specific capabilities to a general-purpose processor in a single chip, the NetVM adds specific functionalities for network processing to a standard stack-based instruction set. We call these specific functionalities *coprocessors*. The advantage of providing this is twofold. First, they can be mapped on features (where present) of real network processors, exploiting the advanced hardware and the coprocessors that often equip them. This allows to greatly increasing the efficiency of the programs when the target platform provides the proper hardware. For example, a program using the CRC32 functionality of a CRC coprocessor will be very efficient and simple on a platform with CRC hardware. Second, on general purpose systems they make use of optimized algorithms. They share code and data structures among different modules and thus grant good resource usage. For example, in a configuration with several NetPEs that call the CRC32 functionality, the same coprocessor can be used by all these NetPEs. Furthermore, if the efficiency of the CRC32 is improved with a better algorithm, every NetPE that uses it becomes faster, therefore the whole configuration (an not only a module) is enhanced. Finally, tasks like 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 multiple strings.

NetPEs communicate with coprocessors by means of a well-defined interface. Figure 1 shows (at the bottom) some coprocessors that may be present in the NetVM reference design.

V. PERFORMANCE EVALUATION

Although the current implementation of the NetVM is still in the early stages, we can figure out some numbers in order to evaluate the goodness of the proposed architecture.

This preliminary performance evaluation is carried out against the BPF, probably the most known virtual machine in network processing arena. Figure 3 shows a very short program in BPF assembly language that, given an Ethernet frame, it checks if the frame contains an IP packet. Figure 4 contains a similar program coded according to the NetVM language.

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

Figure 3. Example of code that filters IPv4 packets with the BPF virtual machine.

```
; Push Port Handler
;   triggered when data is present on a push-input port
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_pkt   ; compare the 2 topmost values; jump if true
  ret                ; otherwise do nothing and return

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

Figure 4. Example of code that filters IPv4 packets with the NetVM virtual machine.

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 PBF 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 III shows the time needed to execute the programs reported in Figure 3 and Figure 4: as expected, the BPF outperforms the NetVM. For instance NetVM is penalized not only due to the additional instructions, but also due to the fact that BPF code is translated into native x86 assembly, while NetVM instructions are interpreted at run-time.

TABLE III
NETVM PERFORMANCE EVALUATION.

Virtual Machine	Time for executing the "IPv4" filter (clock cycles)
NetVM	2236
BPF	124

However, a NetVM is intended as a reference design and its code is not expected 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) 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 a just in time (JIT) compiler is part of our future work in the NetVM.

VI. CONCLUSIONS

This paper presents the architecture Network Virtual Machine (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 performing packet processing 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 processing applications across different hardware and software platforms. Further, the proposed architecture can be used as reference architecture for the implementation of hardware (integrated) networking systems. Finally, the NetVM can be used as a novel tool for specification, fast prototyping, and implementation of hardware (integrated) 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 Just In Time compiler (JITTER) for NetVM code aims at reversing or at least reducing this performance discrepancy.

Since writing NetVM native code (bytecode) is not practical, 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 JITTER for a commercial network processor.

BIBLIOGRAPHY

[1] Lucent Technologies. "PayloadPlus Functional Programming Language." Preliminary Product Brief. Lucent Technologies, Microelectronics Group. April 2000.

[2] Rothfus, Eric J. "The Case for a Classification Language." Agere White Paper. Sept 10, 1999.

[3] Avatria, ADEPT development environment, http://www.avatria.com.

[4] J. Li, F. R. Boyer, and E. M. Aboulhamid, "Retargetable C Compiler for Network Processors," Proceedings of 6th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2002), Orlando, Fl., July 2002.

[5] C. W. Fraser, D. Hanson, A Retargetable C Compiler: Design and Implementation. The Benjamin/Cummings Publishing Company, Inc. 1994.

[6] Gigascale System Research Center, MESCAL: "Modern Embedded Systems: Compilers, Architectures, and Languages", http://www.gigascale.org/mescal.

[7] Karen Bartleson, A New Standard for System-Level Design, Datasheet, 1999, http://www.systemc.org/.

[8] T. Lindholm, F. Yellin, The Java Virtual Machine Specification Second Edition, 1999.

[9] Steven Perbenton and Martin C. Daniels, Pascal implementation of the P4 compiler, Ellis Horwood, 1982.

[10] ECMA standard 335, Common Language Infrastructure (CLI), 2nd edition, December 2002.

[11] 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.

[12] J.T. Moore, M. Hicks, S. Nettles, Practical Programmable Packets, in Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communication Societies (INFO-COM'01), April 2001.

[13] A. Kind, R. Pletka, B. Stiller, The Potential of Just-in-Time Compilation in Active Networks based on Network Processors, in Proceedings of IEEE OPENARCH'02, June 2002.

[14] F. Risso, L. Degioanni, An Architecture for High Performance Network Analysis. Proceedings of the 6th IEEE Symposium on Computers and Communications (ISCC 2001), Hammamet, Tunisia, July 2001. WinPcap is available at http://winpcap.polito.it/.

[15] 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/.

[16] [32] G. Memik, W. H. Mangione-Smith, NEPAL: A Framework for Efficiently Structuring Applications for Network Processors, Second Workshop on Network Processors – NP2, 2003.

[17] P. Russell et al., Netfilter, http://www.netfilter.org.

[18] S. Karlin, L. Peterson, VERA: an extensible router architecture, in Computer Networks, volume 38, number 3, 2002.

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

[20] B. Chen, R. Morris, Flexible Control of Parallelism in a Multiprocessor PC Router, in Proceedings of the 2001 USENIX Annual Technical Conference, 2001.

[21] 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.