FPL-3E: towards language support for reconfigurable packet processing

Mihai Lucian Cristea[1] and Claudiu Zissulescu[1] and Ed Deprettere[1] and Herbert Bos[2]

[1] {cristea,claus,edd}@liacs.nl, Leiden University, The Netherlands
[2] herbertb@cs.vu.nl, Vrije Universiteit Amsterdam, The Netherlands

Abstract. The FPL-3E packet filtering language incorporates explicit support for reconfigurable hardware into the language. FPL-3E supports not only generic header-based filtering, but also more demanding tasks such as payload scanning and packet replication. By automatically instantiating of hardware units (based on a heuristic evaluation) to process the incoming traffic in real-time, the NIC-FLEX network monitoring architecture facilitates very high speed packet processing. Results show that NIC-FLEX can perform complex processing at gigabit speeds. The proposed framework can be used to execute such diverse tasks as load balancing, traffic monitoring, firewalling and intrusion detection directly at the critical high-bandwidth links (e.g., in enterprise gateways).

Key words: High-speed packet processing, reconfigurable hardware, network monitoring

1 Introduction

There exists a widening gap between advances in network speeds and those in bus, memory and processor speeds. This makes it ever more difficult to process packets at line rate. At the same time, we see that demand for packet processing tasks such as network monitoring, intrusion detection and firewalling is growing. Commodity hardware is not able to process packet data at backbone speeds, a situation that is likely to get worse rather than better in the future. Therefore, more efficient and scalable packet processing solutions are needed.

It has been recognised that parallelism can be exploited to deal with processing at high speeds. A network processor (NP), for example, is a device specifically designed for packet processing at high speeds by sharing the workload between a number of independent RISC processors. However, for very demanding applications (e.g., payload scanning for worm signatures) more power is needed than any one processor can offer. For reasons of cost-efficiency it is infeasible to develop NPs that can cope with backbone link rates for such applications. An attractive alternative is to use a reconfigurable platform such as an FPGA that exploits parallelism at a coarser granularity.

We have previously introduced the efficient monitoring framework Fairly Fast Packet Filters (FFPF) [2], that can reach high speeds by pushing as much of the work as possible to the lowest levels of the processing stack (see Fig. 1.b). The NIC-FIX architecture [11] showed how this monitoring framework could be extended all the way down

to the network card. To support such an extensible programmable environment, we introduced the special purpose FPL-3 language.

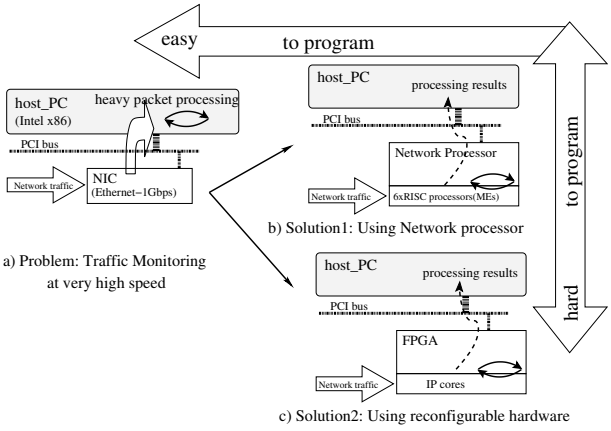

Fig. 1. Moving to special purpose embedded systems.

In this paper, we exploit packet processing parallelism at the level of individual processing units (FPGA cores) to build a monitoring architecture: NIC-FLEX(see Fig. 1.c). Incoming traffic is stored into a fast off-chip memory, wherefrom it is processed by multiple FPGA cores in parallel. The processing results are first stored into a very fast local memory and then passed, on demand, to a higher level (e.g., user space tools). The main contribution of this paper consists of a novel language that explicitly facilitates parallelisation of complex packet processing tasks: FPL-3E. Also, with NIC-FLEX we extend the FFPF architecture upwards with specific packet processing support to create a flexible and fast filtering platform. Experiments show NIC-FLEX to be able to handle complex tasks at gigabit line-rate.

This paper builds on the idea of extensible system-on-programable-chip that was advocated by Lockwood *et al.* in [8] for firewalling. However, we use it to provide a generic high-speed packet processing environment by using the Compaan/Laura tool chain [7, 15] that automatically transform a user code into synthesizable VHDL code that targets a specific FPGA platform.

The remainder of this paper is organised as follows. In Section 2, the architecture of the packet processing system and its supporting language are presented. Section 3 is devoted to the implementation details. The proposed architecture is evaluated in Section 4. Related work is discussed throughout the text and summarised in Section 5. Finally, conclusions are drawn and options for future research are presented in Section 6.

2 Architecture

2.1 High-level overview

At present, high speed network packet processing solutions need to be based on special purpose hardware such as dedicated ASIC boards or network processors (see Fig. 1.b).

Although faster than commodity hardware (see Fig. 1.a), solutions based even on these platforms are surpassed by the advances in reconfigurable hardware systems, e.g., FPGAs.

To counter this scalability trend we propose the solution shown in Fig. 1c, which consists of mapping the user's program onto hardware, processing the incoming traffic efficiently, and then passing the processing results back to the user.

The software architecture composes by three main components (see Fig. 2). The first component ① is a high level interface to the user and the kernel space of an Operating System (e.g., Linux) and is based on the Fairly Fast Packet Filter (FFPF) [2] framework. The second component ② is the FPL-compiler interface between the first and the last components. The compiler takes a program written in a packet processing language ⓐ and generates a code object ⓑ for the lowest level of processing: Reconfigurable hardware. The third component ③ is a synthesiser tool that maps specific processing algorithms onto an FPGA platform and is based on Laura tool.

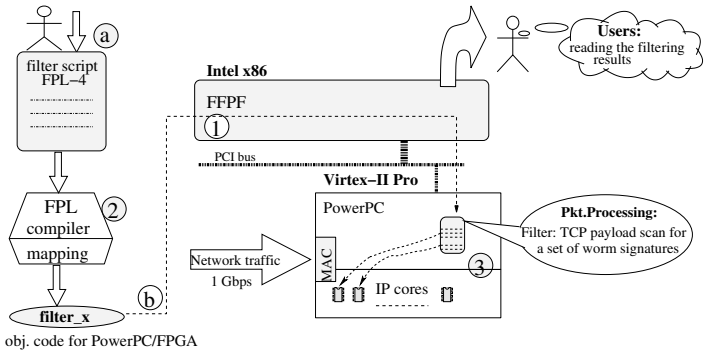

Fig. 2. Packet processing architecture.

2.2 The FFPF software framework

FFPF was designed to meet the following challenges: (1) monitor high-speed links and scale with future link rates, (2) offer more flexibility than existing packet filters, and (3) provide a migration path by being backward compatible with existing approaches (notably `pcap`-based applications [10]). The FFPF framework supports userspace programs, kernel, the IXP1200 network processor, or a combination of the above. FFPF now extends also on reconfigurable hardware by introducing explicitly support on FPGA through its FPL-compiler extensions enclosed into the FFPF programming language (FPL-3E).

2.3 The FPL-3E language and the FPL-compiler

As our architectural design relies on explicit hardware support, we needed to introduce this functionality into our framework. With FPL-3E, we adopted a language-based approach, following our earlier experiences in this field. We designed FPL-3E specifically with these observations in mind: First, there is a need for executing tasks (e.g., payload

scanning) that existing packet languages like BPF [10], Snort [13] or Windmill [9] cannot perform. Second, special purpose devices such as network processors or FPGAs can be quite complex and thus are not easy to program directly. Third, we should facilitate on-demand extensions, for instance through hardware assisted functions. Finally, security issues such as user authorisation and resource constraints should be handled effectively. The previous version of the FPL-3E language, FPL-3 [6], addressed many of these concerns. However, it lacked features fundamental to reconfigurable hardware processing like resource partition and parallel processing.

We will introduce the language design with an example. First, a simple program requiring a high amount of processing power is introduced in Figure 3. Then, the same example is discussed through multiple 'mapping' cases by using the FPL-3E language extensions in Figures 4, 5.

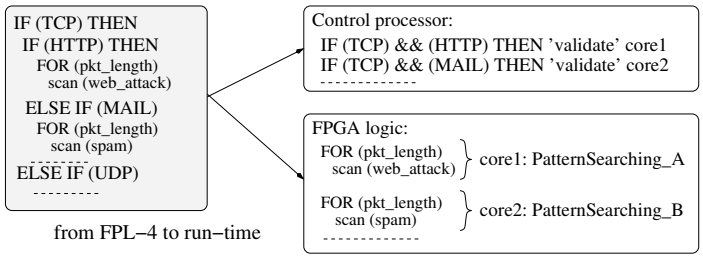

Fig. 3. Packet processing example.

As Figure 3 shows, the FPL-3E compiler translates the program into multiple output objects, one for a control processor (ASIC embedded into the FPGA) and a second one for the FPGA reconfigurable hardware (logic that contains multiple cores). The FPGA cores consist of specific pattern searching algorithm implementations (e.g., regular expressions, Aho-corasick) that are interconnected in such way as to achieving an optimal processing path as we show later in this section. Besides the parallelism built into the logic, we note that the task from embedded control processor runs itself in parallel with the FPGA logic. The control code is mostly composed of nested IF statements used for result validation and, therefore, the processing speed of the control processor is high enough to catch up with the high speed FPGA data processing.

Note that the requirement to perform complex packet processing at the Gbps line rate means that each packet has to be processed within a very limited time budget – a basic task. When a task requires a large amount of *per-packet* processing power (e.g., a full packet scan for a worm), it becomes infeasible to perform this task on a single processing unit when network speeds go up. Thus, we give the same example mapped using various techniques for parallel processing environment. For the sake of simplicity we limit our granularity to three levels.

A basic processing task consists in searching through the whole packet payload data for a string (e.g., a worm signature) and it is performed by a processing unit implemented in hardware. When the task overloads the processing unit, then this task can be distributed across three hardware units in parallel, using one search key per packet, as shown in Figure 4.a, or multiple keys per packet (see Fig. 4.b), or a combination of both

techniques. In the first configuration, the required number of cycles is reduced with the number of hardware devices instantiated – three in our example, as the same string is searched on different parts of the packet. The second approach allows us to search in parallel three signatures on the same packet at a cycles cost of one. However, when the receiving rate is higher than the processing abilities given by 'one packet' approach, we can process multiple packets in parallel (depth-processing), as illustrated in Figure 5.

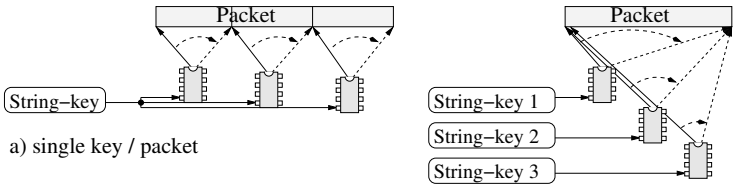

Fig. 4. Packet processing techniques.

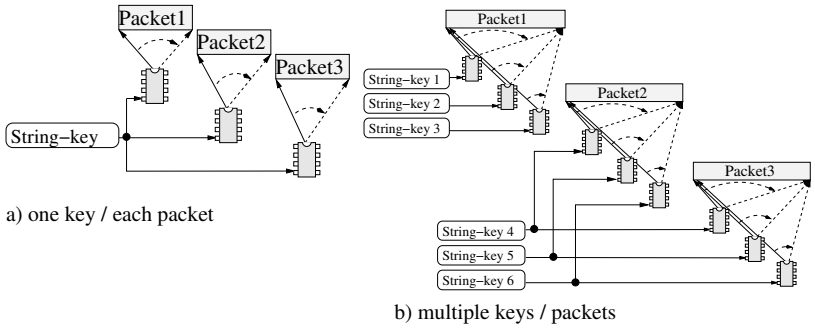

Fig. 5. Multi-Packet processing techniques.

The FPGA technology gives us enough flexibility to choose for one or a mix of the above mentioned approaches. It also provides a long-term platform life by its ease-to-extend with new algorithm implementations, such as IP cores specifically designed for pattern matching, regular expressions, protocol recognition, etc. This support will address future issues like adaptivity to new protocols (e.g., peer-to-peer). The limitation is given only by the hardware capacity (that nowadays goes beyond our needs) and the compiler abilities to perform such complex mapping from a simple and 'natural' programming language: FPL-3E.

2.4 The Compaan/Laura tool chain

The FPGA platform is a highly parallel structure suitable to accommodate algorithms that exploit this parallelism. Although this texture is the key to take advantage of the platform, the commonly used imperative specification programming languages like C, Java or Matlab are hard to compile to parallel FPGA specifications. In general, specifying an application in a parallel manner is a difficult task. Therefore, we used the

Compaan Compiler [7] that fully automates the transformation of sequential specification to an input/output equivalent parallel specification expressed in terms of a so-called Kahn Process Network (KPN). Subsequently, the *Laura* tool [15] takes as its input this KPN specification and generates synthesizable VHDL code that targets a specific FPGA platform. The Compaan and Laura tools together realise a fully automated design flow that maps sequential algorithms onto a reconfigurable platform. We use this design tool chain to implement our computational intensive cores such as pattern matching algorithms. Thus, in FPL-3E, we separate the control intensive tasks from data intensive tasks. These last tasks are automatically analysed and mapped onto an FPGA platform thereby exploiting the inherit parallelism of the data processing algorithms.

3 Implementation details

3.1 The FPL-3E language

The FFPF programming language (FPL) was devised to give the FFPF platform a more expressive packet processing language than previously available. The FPL-3E syntax is summarised in Figure 6. It supports all common integer types and allows expressions to access any field in the packet header or payload in a friendly manner.

The latest version (FPL-3) conceptually uses a register-based virtual machine, but compiles to fully optimised object code. FPL-3 supports commodity PCs and NPs (further implementation details available in [6]). We now introduce its direct descendant, FPL-3E, which extends FPL-3 with constructs for reconfigurable hardware processing.

EXTERN() construct. This was introduced in FPL-3 to support the 'extensibility' system feature. We extend it with support for reconfigurable hardware devices (FPGAs). `EXTERN(name,input,output,hw_depth)` tells the compiler that the task needs the help of the specified core 'name' to process the current packet according to 'input' parameters and place the processing results in the 'output'. 'hw_depth' is an optional parameter for advance users that want to 'force' the compiler to use a certain amount of hardware units for parallel packet processing. By default, the compiler estimates this parameter accordingly to the incoming traffic rate and the available hardware resources (given at compile time).

3.2 The FPL-3E compiler

The FPL-3E source-to-source compiler, like its predecessors, generates straight C target code that can be further handled by any C compiler. Programs can therefore benefit from the advanced optimisers in the Intel μEngine C compiler for IXP devices, `gcc` for commodity PCs and Xilinx ISE for Xilinx's FPGAs. As a result, the object code will be heavily optimised even though we did not write an optimiser ourselves.

Moreover, the FPL-3E compiler uses a heuristic evaluation of the hardware instances needed to reach the system goal (e.g., the line rate is 1Gbps). The evaluation is based on the workload given by one hardware instance to perform the user's program and a critical point where the performances fell down because of some heavy computation like signature length, or packet size. For example, in Figure 5.b, assuming that

the user's program performs checking of six signatures, but three of them are known as much longer than the others, the compiler duplicates the hardware units, respectively, in order to achieve a well balanced workload of the whole system.

operator-type	**operator**
Arithmetic	`+, -, /, *, %, --, ++`
Assignment	`=, *=, /=, %=, +=, -= <<=, >>=, &=, ^=, |=`
Logical / Relational	`==, !=, >, <, >=, <=, &&, ||, !`
Bitwise	`&, |, ^, <<, >>`

statement-type	**operator**
if/then/else	IF (expr) THEN stmt1 FI ELSE stmt2 FI
for()	FOR (initialise; test; update) stmts; BREAK; stmts; ROF
return a value	RETURN (val)
external function	INT EXTERN(name,input, output) or INT EXTERN(name,input, output,hw_depth)

Data type	**syntax**
Register n	R[n]
Memory location n	M[n]
Packets access:	
-byte $f(n)$	PKT.B[$f(n)$]
-word $f(n)$	PKT.W[$f(n)$]
-double word $f(n)$	PKT.DW[$f(n)$]
-bit m in byte n	PKT.B[n].U1[m]
-nibble m in byte n	PKT.B[n].U4[m]
-bit m in word n	PKT.W[n].U1[m]
-byte m in word n	PKT.W[n].U8[m]
-bit m in dword n	PKT.DW[n].U1[m]
-byte m in dword n	PKT.DW[n].U8[m]
-word m in dword n	PKT.DW[n].U16[m]
-macro	PKT.macro_name
-ip proto	PKT.IP_PROTO
-ip length	PKT.IP_LEN
-etc.	customised macros

Fig. 6. FPL-3E language constructs

3.3 Control processor and FPGA cores

In today FPGAs there are embedded from one to four hard cores control processors (e.g., PowerPC or ARM), that are suitable to map the control part of our algorithms. The data intensive task are mapped directly in hardware (IP cores) using the Compaan/Laura tool chain. The IP cores communicate with the control processor using a set of registers to set some run-time parameters (e.g., the packet length or the searched string).

To study the feasibility of using the Compaan/Laura tool chain in the Networking world we compiled in hardware a searching algorithm. The Matlab program for this algorithm is shown in Figure 7. The bytes of the packet(e.g., $pkt()$) are compared with the content of a signature string (e.g., $sig()$). If the signature is present in the packet, then the value of the *c* variable is equal with the length of the searched string.

The program illustrated in Figure 7 has been rewritten to match the requirements of the Compaan/Laura tool chain. Additionally, we instructed our tool to generate a design that compare eight characters in parallel. The hardware network of processors is depicted in Figure 8. Each bubble represents a hardware processor and each arch a communication channel between two processors. The *ReadPacket* processor feeds our network with packet bytes from a MAC network interface. The *Search* processors implements the character wise searching, the result of a searching of a string is evaluated by the *Eval* processor, which is also our write interface with external devices.

Table 1 gives the hardware results of the FPGA implementation of the algorithm given in Figure 7. The experiment has been done using Symplify and ISE Xilinx 6.2 for the Virtex II-6000 platform. The hardware is capable to do an eight character string

search in a variable packet size. The required number of cycles for a variable packet size and eight characters search string is $cycles = 13 + PackSize$.

```
for i = 7: 1: PackSize,
  c = 0;
  for j = 1: 1: StringLength,
    if sig(j) = pkt(i),
       c = c + 1;
    end
  end
  if c = StringLength,
    print "Found!"
  end
end
```

Fig. 7. Simple Search Algorithm

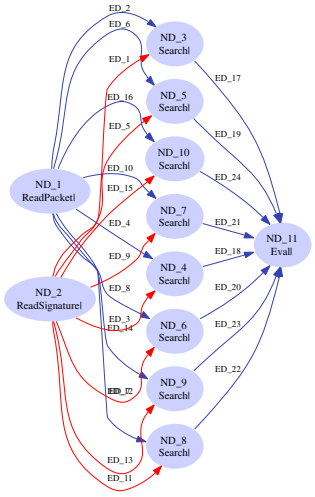

Fig. 8. Processor Network of the simple algorithm

In our example, the length of the search string is fixed to eight characters. However, the string content may be changed at run-time.

Packet Length	String Length	Clocks/Workload	Slices	Frequency (MHz)
64	8	77	2035	101

Table 1. Experimental results

4 Evaluation

Given the pattern matching algorithm result (see Table 1) for one search-key per packet, we extrapolate to other case studies as already illustrated in Figure 5. In Figure 9 is shown how the performances of one key per packet approach (1key/1hw) scale up by increasing the use of hardware units (1key/3hw) in parallel.

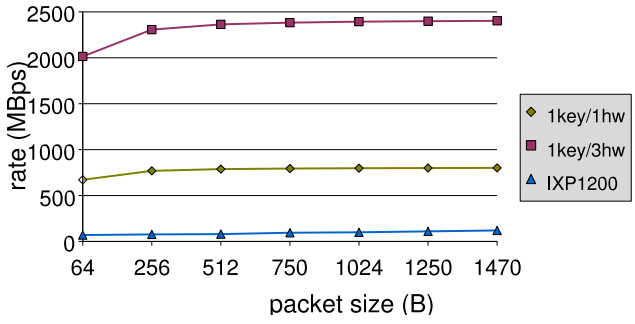

Fig. 9. FPGA vs. NP processing results.

The processing result of a full packet payload pattern search filter performed by a 1Gbps generation network processor (Intel IXP1200) is also shown in Figure 9. Therefore, making a comparison between an FPGA implementation and a network processor implementation, it can be seen that a complex filter (such as a pattern searching algorithm) performed by a NP is surpassed by even a single IP core implementation.

Note that the relative small amount of hardware resources used for this implementation (ca. 6% for a Virtex II-6000) allows us to map more than one search engine into a FPGA platform.

5 Related work

The usage of accelerated cores has been done in the Molen project [14] by annotating a PowerPC processor with a set of multimedia instructions that are accelerated in hardware. Our approach focuses on the networking applications algorithms. Thus, our hardware accelerated cores perform coarse grain computations (e.g., pattern matching). Additionally, the number of processing elements for a particular task can be set-up at compile time based either on user demands or on the built-in compiler heuristic estimator about the workload requirements.

Using reconfigurable hardware for increased packet processing efficiency was previously explored in [4] and [1]. Our architecture differs in that it provides explicit language support for this purpose. As shown in [3], it is efficient to use a source-to-source compiler from a generic language (Snort Intrusion Detection System) to a back-end language supported by the targeted hardware compiler (e.g., Intel μEngineC, PowerPC C, VHDL). We propose a more flexible and easy to use language as front-end for users. Moreover, our FPL-3E language is designed and implemented for heterogeneous targets in a multi-level system.

The SCAMPI architecture also pushes processing to the NIC [12]. It assumes that hardware can write packets immediately into host memory (e.g., by using DAG cards [5]) and implements access to packet buffers through a userspace daemon. SCAMPI does not support user-provided external functions, powerful languages such as FPL-3E.

6 Conclusions and future work

This paper presented the NIC-FLEX packet processing environment and its FPL-3E programming language, which enable users to process network traffic at high speeds by mapping of their programs onto reconfigurable hardware (FPGA). A program is mapped by loading IP cores generated using Compaan/Laura approach to implement the data intensive tasks in hardware from the real of networking. Currently, this task is performed by an engineer and thus, the user cannot generate its own tasks in hardware. However, we supply the FPL framework with a wide range of hardware cores to overcome the need for different cores. All these cores are annotated with performace numbers such that the FPL-3E environment computes the right work balance, based on a heuristic evaluation. This heuristics may be ignored by the user and replaced with its own evaluation. The experimental results show that NIC-FLEX can outperform traditional packet filters by processing at Gbps linerate.

In the future, we plan to extend NIC-FLEX with a management environment that can take care of object code loading and program instantiation.

Acknowledgements

This work was supported by the EU SCAMPI project IST-2001-32404, while Intel donated the network cards and Xilinx donated the development kit.

References

1. D. Anto, J. Koenek, K. Minakov, and V. ehk. Packet header matching in combo6 ipv6 router. Technical Report 1, CESNET, 2003.
2. H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis. FFPF: Fairly Fast Packet Filters. In *Proceedings of OSDI'04*, San Francisco, CA, December 2004.
3. I. Charitakis, D. Pnevmatikatos, and E. Markatos. Code generation for packet header intrusion analysis on the ixp1200 network processor. In *SCOPES 7th International Workshop*, 2003.
4. C. Clark, W. Lee, D. Schimmel, D. Contis, M. Kone, and A. Thomas. A hardware platform for network intrusion detection and prevention. In *The 3rd Workshop on Network Processors and Applications (NP3)*, Madrid, Spain, Feb 2004.
5. J. Cleary, S. Donnelly, I. Graham, A. McGregor, and M. Pearson. Design principles for accurate passive measurement. In *Proceedings of PAM*, Hamilton, New Zealand, Apr. 2000.
6. M. L. Cristea, W. de Bruijn, and H. Bos. Fpl-3: towards language support for distributed packet processing. In *Proceedings of IFIP Networking 2005 (accepted for publication)*, Waterloo, Canada, May 2005.
7. B. Kienhuis, E. Rypkema, and E. Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *Proceedings of the 8th International Workshop on Hardware/Software Codesign (CODES)*, San Diego, USA, May 2000.
8. J. W. Lockwood, C. Neely, C. Zuver, J. Moscola, S. Dharmapurikar, and D. Lim. An extensible, system-on-programmable-chip, content-aware Internet firewall. In *Field Programmable Logic and Applications (FPL)*, page 14B, Lisbon, Portugal, Sept. 2003.
9. G. R. Malan and F. Jahanian. An extensible probe architecture for network protocol performance measurement. In *Computer Communication Review, ACM SIGCOMM*, Oct. 1998.
10. S. McCanne and V. Jacobson. The BSD Packet Filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 Winter USENIX conference*, San Diego, Ca., Jan.
11. T. Nguyen, W. de Bruijn, M. Cristea, and H. Bos. Scalable network monitors for high-speed links: a bottom-up approach. In *Proceedings of IPOM'04*, Beijing, China, 2004.
12. M. Polychronakis, E. Markatos, K. Anagnostakis, and A. Oslebo. Design of an application programming interface for ip network monitoring. In *IEEE/IFIP NOMS*, Seoul, April 2004.
13. M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA Systems Adminstration Conference*, 1999.
14. S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte. The molen polymorphic processor. *IEEE Transactions on Computers*, November 2004.
15. C. Zissulescu, T. Stefanov, B. Kienhuis, and E. Deprettere. LAURA: Leiden Architecture Research and Exploration Tool. In *Proc. 13th Int. Conference on Field Programmable Logic and Applications (FPL'03)*, Lisbon, Portugal, Sept. 1-3 2003.