Modeling filtering predicates composition with Finite State Automata

Modeling Filtering Predicates Composition with Finite State Automata

Marco Leogrande, Luigi Ciminiera, Fulvio Risso
Dipartimento di Automatica e Informatica
Politecnico di Torino
10129 Torino, Italy
Email: `$first.$last@polito.it`

Abstract—**Designing an efficient and scalable packet filter for modern computer networks becomes each day more challenging: the increased data rate and a much higher number of active hosts make filtering expressions more complex than in the past, noticeably stressing the filtering systems. Some architectures focus on the most used filtering patterns and predicates, but must use a slower fallback if filtering on "less used" ones; dynamic code generation systems implement complex optimizations, but these compiler-oriented algorithms focus only on the final code, where most of the protocol abstraction is lost. This paper presents mpFSA, a novel packet filter model based on the Finite State Automata formalism to guarantee optimality w.r.t. the number of packet accesses, that aims at allowing powerful optimizations without sacrificing efficiency and scalability.**

I. Introduction

In the rapidly growing world of network connections, packet filtering becomes each day more challenging. On one hand, many more data are moved around, because either the requested service is bandwidth-eager (file sharing, multimedia content streaming) or it is run in the cloud, therefore requiring more network interaction with the user client. On the other hand, the number of handheld (or similar) devices connected to the Internet is steadily increasing; even if the traffic generated and requested by them is relatively small, they appear as active speakers on the network.

This situation puts packet filters under much more stress, because of the increased data rate and number of flows. Therefore, the research focuses on developing efficient and scalable packet filters, that, at the same time, offer good filtering precision and modularity.

There are many filtering implementations that strive for efficiency, each one applying its own set of optimizations, both in hardware and software. All of them, however, implement ah-hoc optimizations: to the best of our knowledge, there is no approach that aims at solving the problem with a unified formalism. In particular, with current systems it is not possible to satisfy all of these conditions: *(i)* perform multiple checks at the same time (as when filtering session traffic); *(ii)* analyze less common filtering patterns without a decrease in matching speed; *(iii)* combine totally different filters.

Furthermore, in many tools the protocol definition database is hard-coded and therefore cannot be easily extended or modified to recognize custom or hand-made protocols.

This paper proposes a new model, called **mpFSA (Multilevel Finite State Automata with Predicates)**, that aims at defining a unified approach to the filtering optimizations, instead of ad hoc multiple optimizations techniques, guaranteeing efficiency and scalability both in the common and in the corner cases. The model is focused on the reduction of the number of packet accesses, even if the filtering predicate is complex or is expressed as a function of seldom used protocol fields. This model is generic enough not to require *a priori* protocol definitions; in our implementation, in fact, we use a flexible protocol database known only at runtime. No assumptions are made about the context in which the model is used: optimizations aim at scanning each packet as efficiently as possible, but no further application-level decisions are taken (i.e. conflicting firewall rules resolution).

This paper is structured as follows. Section II presents the state of the art; Section III introduces the proposed model; Section IV describes our experimental evaluation; Section V draws the conclusions.

II. Related Work

The **CMU/Stanford Packet Filter** [1] (or, shortly, **CSPF**) implementation was developed around 1987 and has been a pioneer in the packet filtering field. This system introduced the concept of a kernel-level virtual machine inside which an application-provided program runs; if that code returns *true*, the packet is forwarded to the application.

The **Berkeley Packet Filter** [2] (often shortened in **BPF**), was created to overcome the limitation of the previous architectures and it is based on a more powerful virtual machine. The most important improvement of this architecture is the adoption of the *Control Flow Graph* model, that permits compiler techniques to be used to remove redundant checks from the generated code.

PathFinder [3] extends the coalescing technique proposed by BPF, compacting the Control Flow Graphs related to different filters. It uses a *trie* as the data structure to hold the data; each atomic unit of the trie is called *cell*. Whenever a user submits a filter to PathFinder, its expression is exploded in a list of cells, each one describing a step in the construction of the final check. When a filter is expressed on different predicates or protocol fields, PathFinder tries hard to maximize the number of common cells, therefore allowing a better compaction. On the other hand, filters are optimized only if they share a common prefix[1]. **DPF** [4] (acronym for **Dynamic Packet Filter**) improves this approach: instead of emitting VM code to be interpreted at packet receipt time, it creates machine-specific code (that can be run natively) each time a filter is added or removed. Furthermore, memory access is improved, because byte checks at contiguous offsets are performed concurrently.

BPF+ [5] improves the BPF model, keeping its VM approach, but using much more aggressive optimizations and a JIT compiler; it is considered state-of-the-art with regard to filter optimization. Some notable optimizations are: *(i) Redundant Predicate Elimination*, aiming at removing redundant checks on the same protocol field; *(ii) Partial Redundancy Elimination*, that removes multiple loads from memory of the same packet data; *(iii)* the *Lookup Table Encapsulation*, instead, optimizes multiple checks on the same field in a single hash table access.

Swift [6] was presented as a "packet filter for high performance packet capture on commercial off-the-shelf hardware". Actually its designers focused their attention on the filter update times and on the creation of a new virtual machine. The Swift VM uses a tree-like structure, like PathFinder did, and exploits the native hardware possibilities by enriching the instruction set with *SIMD* instructions, capable of executing multiple checks in parallel.

SPAF [7], acronym for **Stateless FSA-based Packet Filters**, is a recent proposal that aims at using Finite State Automata as well. A database containing protocol specifications (both protocol fields' format and their encapsulation rules) is scanned and an automaton is created for each of them, that reads the bytes that are part the protocol and follows the encapsulation rules (i.e. the start state of the `IP` protocol is linked from the exit state, among those of the `Ethernet` protocol, that is reached when the bytes that compose the `EtherType` have the proper value). The different automata are then joined together using the algorithms from the literature. The major drawback of SPAF is that the protocol field abstraction is lost very early in the computation, as the generated automata read directly the packet bytes (differently from the mpFSA approach, described in Section III); therefore much more transitions are created than necessary.

Other technologies from the state of the art, as the **FFPF** [8] architecture, are not described in detail here, as they aim at solving practical problems (i.e. how to multiplex incoming packets between different filtering implementations), but do not offer any improvement to the filtering model itself.

To the best of our knowledge, the most common filtering architectures apply their algorithms directly to the code that has to be run. The approaches presented above, in fact, aim at speeding up the execution by applying compiler-oriented optimizations to the generated code (either if it is run inside a Virtual Machine or if native instructions are compiled *Just In Time*); some of them also deploy optimizations to coalesce packet accesses or use hardware-efficient assembly instructions. The **mpFSA** proposal aims instead at defining a formalism to optimize directly the *filtering graph* (the data structure that is later used by the compiler to emit the desired code), in a scalable and efficient manner, without sacrificing system modularity and filtering precision.

III. Multilevel Finite State Automata with Predicates

The **mpFSA** architecture proposed in this paper extends the Finite State Automata model, adapting it to packet filtering. The advantage is that a well-defined algebra exists already for FSA, that allows their composition (union, intersection, negation) with solid guarantees of optimality.

A. mpFSA definition

A **Multilevel Finite State Automata with Predicates** (or, briefly, **mpFSA**) is described succinctly in the "five-tuple" notation, very similar to the classic one for FSA [14]:

$$A = (Q,\ \Sigma,\ \delta_p,\ q_o,\ F)$$

where:

Q is a finite set of *states*.
Σ is the set of *input symbols*.
$\mathbf{\delta_p}$ is a *transition function augmented with predicates*, described below.
$\mathbf{q_0}$ is the *start state*, among those in Q.
F is a set of *accepting states*, among those in Q.

A **transition function augmented with predicates** mimics the meaning of "classic" transitions, but adds the possibility to tune the transition behavior according to a set of **Boolean predicates**, whose semantic is orthogonal to the one of input symbols. It is defined as a function:

$$\delta_p(q_1,\ a,\ P) = Q_2$$

where:

$\mathbf{q_1}$ is the state from which the transition takes place.
a is the input symbol on the reception of which the transition fires, or the special value ϵ (*epsilon*) if no input symbol should be consumed.
P is a set of Boolean predicates that drive, internally, the transition. According to the actual Boolean value of these predicates, the transition might behave differently. If $\mathbf{P}$ is empty, the transition is in fact equivalent to a "classic" transition.
$\mathbf{Q_2}$ is the set of states reached by the current transition. The actual states reached depend on the actual Boolean values of the predicates in $\mathbf{P}$.

Figure 1 depicts an example of a very simple mpFSA. When the control is in the `start` state, if the symbol `2` is received, then the transition with predicates is activated and, according to the Boolean value of some predicates (not shown in the picture), control goes either to state `A` or to state `C`.

The whole model is called **multilevel** because, from the mpFSA point of view, each transition is a "black box",

[1] In the expression `(tcp.sport == X and tcp.dport == W) or (ip.src == K and tcp.sport == Y)`, for instance, `tcp.sport` is always checked twice.

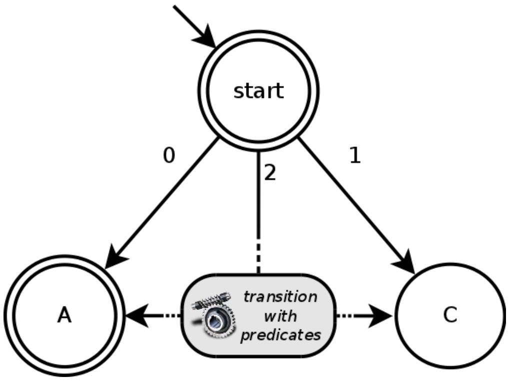

Figure 1. mpFSA example.

whose internal behavior cannot be analyzed; it is necessary to "descend" into it (Section III-D), at the "predicate level", to understand how each transition works exactly.

It is also worth noting that, while input symbols belong to the input alphabet $\mathbf{\Sigma}$ and are a well-known list of characters that are consumed sequentially by the mpFSA (as it is for classic FSA), the predicates live in a completely different space: they can be seen as a set of hypotheses that can be either *true* or *false* (a Boolean value). Note that no assumptions are made on the predicates: in particular, their values are not guaranteed to be constant over time.

The model described so far has no evident relationship with packet filtering; this is indeed true, as the "bare" mpFSA model can be used in any context where it may be useful. In the following subsections the usage for packet filtering will be detailed, by describing the meaning of the input symbols and predicates with regards to packet filtering. An overview of the system is given in Figure 2.

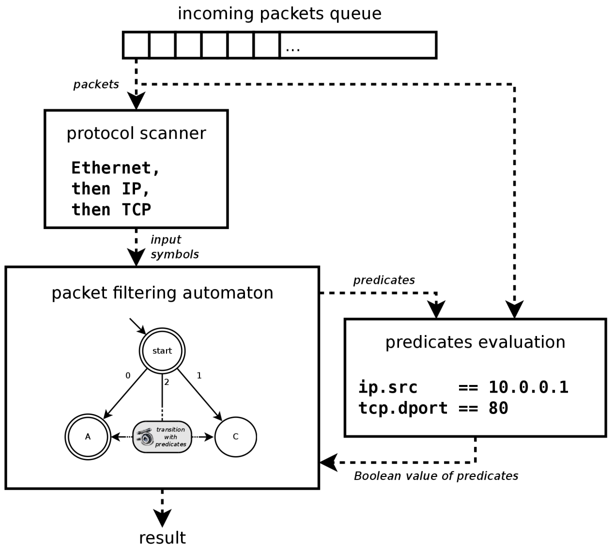

Figure 2. Overview of the system in which the mpFSA model runs.

B. Input symbols

The mpFSA model optimizes the protocol scan by mapping the protocol encapsulation on FSA *input symbols*. Each symbol consumed by a mpFSA represents a "jump" between two consecutive protocols in the packet under investigation: so the symbol `Ethernet-to-IP` carries the meaning that the `IP` protocol was found inside the `Ethernet` encapsulation. Input symbols are generated by an upstream module, that is called *protocol scanner* in Figure 2.

As a consequence, each state of the mpFSA represents the protocol that has been reached while scanning the current packet. As an example, consider the mpFSA in Figure 3, that models the filter `ip.dst == 1 or ip.src == 2 or ip.src == 3`. Control begins in the `begin` state; if the first scanned protocol is `Ethernet`, control moves to the associated state, otherwise the failure state is reached[2]. Then, if an `Ethernet-to-IP` symbol is received while in the `Ethernet` state, control descends into the transition with predicates (described in Section III-D). From the mpFSA point of view, that transition is a black box with two exit paths: `exit_1` and `exit_2`. Which exit paths are enabled or not depends on the predicates that the transition accesses[3].

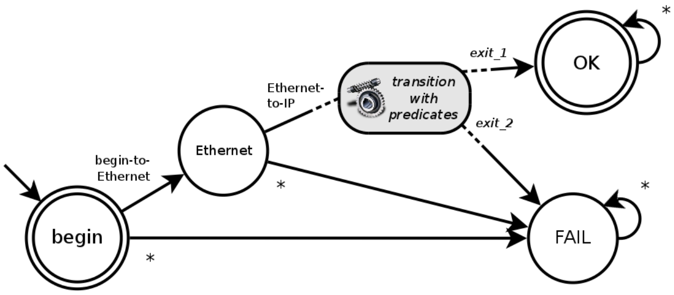

Figure 3. Example of a mpFSA for the filter `ip.dst == 1 or ip.src == 2 or ip.src == 3`. The internal structure of the transition with predicates is detailed later, in Figure 4.

C. Enhancing the transitions: predicates

While filtering packets, predicates are modeled as hypotheses on the fields of the protocols included in the packet itself. The actual Boolean value of the predicates is evaluated by a dedicated module (*predicates evaluation* in Figure 2). This module is logically separated from the protocol scanner, because of the different meaning of input symbols and predicates. Whenever the matching algorithm encounters a predicate, the evaluator is invoked; the current Boolean value of that predicate is returned.

It is worth underlining why no further assumptions are made on the predicates: for instance, the Boolean value

[2] Note that, in Figure 3, the star symbol is used on the transitions with the meaning: "any input symbol that is not handled by other transitions exiting from the same state".

[3] It is worth noting that, in the general case, no assumption is made about the determinism of the mpFSA: there might be a combination of predicates for which both `exit_1` and `exit_2` are enabled.

of `ip.src == 10.0.0.1` is not guaranteed to be constant over time because, if filtering a packet that contains a `IP-in-IP` tunnel, the predicate might have a Boolean value for the external `IP` protocol occurrence and another for the internal one.

D. Going multilevel

We will now focus on the inner workings of the transitions with predicates. It was previously stated that it is necessary to "descend" into the *predicate level*: the formalism used inside the transitions is, indeed, slightly different from the FSA. There is a similarity, though, because there are concepts similar to "states" and "transitions".

The formalism might be explained better with an example: refer to Figure 4, that shows a transition with predicates taken from a mpFSA modeling the filter `ip.dst == 1 or ip.src == 2 or ip.src == 3`. The control "enters" the predicate level from the *input* circle on the left: on the other end, the control can exit from the circles on the right. Moving from the start circle, a field is read from the packet: in this case `ip.dst`. Its value is checked by an operand against some values; in this case, it is checked if it is equal to `1`. If this is the case, the upper path is followed and the transition terminates already; otherwise, the `default` arrow is followed and another check is eventually made. In this case, no more checks are needed on `ip.dst`, so the control goes by reading `ip.src`: if it is equal to `2` or `3`, the same exit state as before is taken, otherwise the bottom one is reached.

As soon as the transition with predicates exits from the "predicate level", it maps, through the associated exit label, the internal exit state that has been reached with the appropriate state in the external mpFSA. The control, then, jumps to that state.

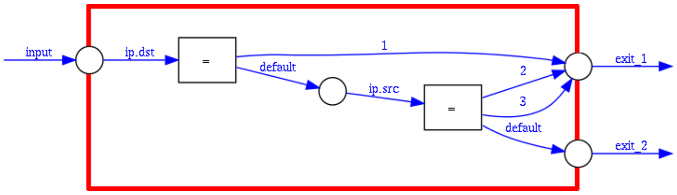

Figure 4. Internal representation of a transition with predicates, part of the mpFSA in Figure 3, that models the filter `ip.dst == 1 or ip.src == 2 or ip.src == 3`. Note the shortcut if the first predicate matches already.

E. Algorithms

One of the main advantages of reusing the FSA formalism is that many definitions, algorithms and optimizations from the literature (for example described in [14]) can be reused with little effort. We were able to concentrate on: *(i)* the efficient mapping on the filters on the FSA model; *(ii)* some small changes in the algorithms, needed because of the finer details of the model, to make them work. For instance, while in the FSA abstraction each transition reaches only a single target state, in the mpFSA formalism transitions with predicates might reach multiple states: this difference required some changes in the composition (union, intersection, negation) and determinization algorithms.

Even if their thorough description is out of the scope of this paper, it is worth noting that only a small amount of time was needed to adapt those algorithms.

IV. Implementation and Validation

We wrote a mpFSA prototype implementation to validate our claims.

The external portion of the model, the one that resembles a Finite State Automaton, was easy to implement, as its characteristics are well-known from the literature already.

The most interesting and challenging part was the implementation of the transitions with predicates and, in particular, the data structure needed inside the *predicate level*. To better decouple the filtering use case from the optimizations that can be made on the predicates, most of the *predicate level* algorithms were split from the main code and implemented in a separate library, that has been called `librange`[4]. This library allows the programmer to define **mappings** between *ranges of arbitrary data* ***keys*** and ***values*** and providing facilities to operate on them (i.e. intersection, traversal). `librange` was then used to manage the mappings between the predicates (as *keys*) and the exit nodes of the mpFSA predicate layer (as *values*).

The proposed mpFSA model have been validated by implementing it inside the NetBee[5] framework, which includes an experimental compiler that creates run-time code for the NetVM [9] virtual machine. The system architecture is described in [10]: the filtering expression, coupled with a protocol database, is given as input to a high-level compiler that generates NetIL code, a NetVM-specific assembly-like language. The generated code can be interpreted by the NetVM itself, for maximum compatibility, or compiled Just-In-Time if a backend compiler is available for the target architecture (Intel x86, Cavium Octeon [11] and Xelerated X11 [12] are currently supported).

In particular, the mpFSA abstraction has been implemented inside the front-end of the aforementioned high-level compiler.

A. Experimental evaluation

In order to validate the mpFSA approach, the prototype has been tested against a BPF implementation, representative of the state of the art. In order to avoid overheads related to the code interpretation, both the mpFSA and the BPF code were compiled Just-In-Time [13].

A packet trace was prepared, that included 300 packets generated during an ordinary web browsing session. Most packets carry HTTP or DNS information, but also packets frequently encountered in small networks were included, like Spanning Tree, ICMP or proprietary L7 protocols.

The following five filtering predicates were chosen:

[4] Code available at: `https://github.com/dark/librange`

[5] More information at: `http://nbee.org/`

1) `ip`
2) `ip src 10.1.1.1`
3) `ip and tcp`
4) `ip src 10.1.1.1 and ip dst 10.2.2.2 and tcp src port 20 and tcp dst port 30`
5) `ip src 10.4.4.4 or ip src 10.3.3.3 or ip src 10.2.2.2 or ip src 10.1.1.1`

All the tests were performed on a single-core machine, a Intel Centrino CPU running at 1.60 GHz, provided with 1 GiB or RAM, booting a Linux OS based on the 2.6.38 kernel, in which the toolchain and the applications were compiled with the version 4.4.5 of `gcc`. A benchmark script was deployed and the number of clock cycles (or *ticks*) needed to complete the packet analysis was measured, by using the `RDTSC` assembly instruction available on the x86 architecture.

To avoid transient issues, the code repeated the filter 100 times for each packet: each packet sample was measured by taking the aggregate run time and dividing it by 100. Furthermore, to eliminate issues related to the profiling algorithm, 10000 runs were executed for each packet, then averaged, by excluding those that were substantially different. These results, computed for each of 300 packets included in the trace, are shown in Figure 5.

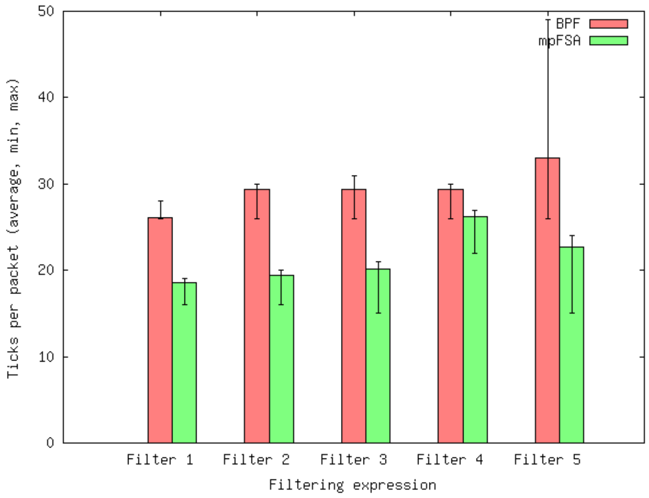

Figure 5. Comparison between BPF and mpFSA performance.

Because of the fewer packet checks needed, caused by the elimination of some redundancies that BPF did not take into account, mpFSA clearly outperforms BPF in all chosen filters. The reduction of the number of ticks needed to complete the filter ranges, in our tests, between 10 and 35 %.

V. Conclusion

This paper presented **mpFSA**, a unified model for the representation and optimization of packet filter complex predicates: the aim is to reduce the number of packet accesses, by using techniques in addition to common compiler optimizations.

A model based on Finite State Automata has been designed, by augmenting the transition function with Boolean predicates. This improvement enables a fine degree of optimizations and an easy filter code generation, because of the model simplicity. A complete system was designed along with the model and implemented as a prototype inside the NetBee framework.

Our preliminary tests show that the performance improvement goal has been achieved. Model optimality is guaranteed by the application of theorems and algorithms already studied in the FSA literature. Our implementation is still far from perfect, but it already performs better than other classic filtering systems. The refinement and formal validation of the implementation, along with a thorough scalability verification, are a challenging task for the future work.

Acknowledgments

The authors would like to thank Olivier Morandi, Lorenzo De Carli and Pierluigi Rolando, who took part in the early stages of this project.

References

[1] J.C. Mogul, R.F. Rashid, M.J. Accetta, The packet filter: An efficient mechanism for user-level network code. In *Proceedings of 11th ACM Symposium on Operating Systems Principles*, Austin, TX, pp. 39-51, Nov. 1987.

[2] S. McCanne, V. Jacobson, The BSD Packet Filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 Winter USENIX Technical Conference*, San Diego, CA, pp. 259-269, Jan. 1993.

[3] M.L. Bayley, B. Gopal, M.A. Pagels, L.L. Peterson, PATHFINDER: A pattern-based packet classifier. In *Proceedings of the First USENIX Symposium in Operating System Design and Implementation*, Monterey, CA, pp. 115-123, Nov. 1994.

[4] D.R. Engler, M.F. Kaashoek, DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of ACM SIGCOMM '96*, Stanford, CA, pp. 53-59, Aug. 1996.

[5] A. Begel, S. McCanne, S.L. Graham, BPF+: exploiting global data-flow optimization in a generalized packet filter architecture. In *SIGCOMM Computer Communication Review*, Vol. 29(4), pp. 123-134, Oct. 1999.

[6] Z. Wu, M. Xie, H. Wang, Swift: a fast dynamic packet filter. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, San Francisco, CA, pp. 279-292, Apr. 2008.

[7] P. Rolando, R. Sisto, F. Risso, SPAF: stateless FSA-based packet filters. In *IEEE/ACM Transactions on Networking (TON)*, Volume 19 Issue 1, Feb. 2011.

[8] H. Bos, M. Cristea, T. Nguyen, G. Portokalidis, FFPF: Fairly Fast Packet Filters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation (OSDI04)*, San Francisco, CA, pp. 347–363, Dec. 2004.

[9] O. Morandi, F. Risso, P. Rolando, S. Valenti, P. Veglia, Creating Portable and Efficient Packet Processing Applications. In *Springer Design Automation for Embedded Systems*, Vol. 15, No. 1, pp. 51-85, March 2011.

[10] O. Morandi, F. Risso, M. Baldi, A. Baldini, Enabling Flexible Packet Filtering Through Dynamic Code Generation. In *Proceedings of IEEE International Conference on Communications (ICC 2008)*, Beijing, China, pp. 5849-5856, May 2008.

[11] O. Morandi, F. Risso, S. Valenti, P. Veglia, Design and Implementation of a Framework for Creating Portable and Efficient Packet Processing Applications. In *Proceedings of the 7th ACM International Conference on Embedded Software (EMSOFT 2008)*, Atlanta, GA, pp. 237-244, Oct. 2008.

[12] O. Morandi, F. Risso, P. Rolando, O. Hagsand, P. Ekdahl, Mapping Packet Processing Applications on a Systolic Array Network Processor. In *IEEE International Workshop on High Performance Switching and Routing (HPSR 2008)*, Shanghai, China, pp. 213-220, May 2008.

[13] L. Degioanni, M. Baldi, F. Risso, G. Varenni, Profiling and optimization of software-based network-analysis applications. In *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing*, Washington, DC, USA, p. 226, 2003.

[14] J.E. Hopcroft, R. Motwani, J.D. Ullman. *Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd Edition, 2006.