Deep Packet Inspection on Commodity Hardware using FastFlow

M. Danelutto, L. Deri, D. De Sensi, M. Torquati
Computer Science Department
University of Pisa, Italy

Abstract. The analysis of packet payload is mandatory for network security and traffic monitoring applications. The computational cost of this activity pushed the industry towards hardware-assisted deep packet inspection (DPI) that have the disadvantage of being more expensive and less flexible.

This paper covers the design and implementation of a new DPI framework using FastFlow, a skeleton-based parallel programming library targeting efficient streaming on multi-core architectures. The experimental results demonstrate the efficiency of the DPI framework proposed, proving the feasibility to perform 10Gbit DPI analysis using modern commodity hardware.

Keywords. Network streaming, NetFlow, DPI, FastFlow, multi-core.

Introduction

When the Internet was designed, all routing and security protocols were conceived to process traffic only based on packet headers. When the original equation TCP/UDP port equal application port could no longer be applied as developers started to implement services over dynamic ports, network monitor companies started to develop packet payload analysis tools aimed at identifying the application protocol being used, or at enforcing laws and copyright.

This trend was further fostered by the introduction of IDS/IPS systems (Intrusion Detection/Prevention Systems) that inspect all packets with the purpose of detecting malicious traffic. These have been the driving forces for the development of DPI (Deep Packet Inspection) tools and frameworks able to inspect packet payload and thus easing the development of applications using them.

Most tools are accelerated using custom hardware network cards such as those based on FPGAs (Field Programmable Gate Array) [1] or proprietary silicon [2] such as chips produced by companies like Cavium, Radisys or Netronome. The use of specialized hardware for accelerating packet inspections and to be sure to fully cope with high-speed network traffic, makes DPI solutions expensive, bound to few vendors, and thus unsuitable for being used for the development of open systems that need to rely on DPI.

Beside the nDPI [3], L7Filter [4] and libprotoident [5] projects, the open-source community does not offer alternatives for software-based freely available DPI tools able to sustain high-speed (10Gbit or more) packet analysis [6].

This work has been partially supported by FP7 STREP ParaPhrase (`www.paraphrase-ict.eu`)

This work aims to design a high-performance, all-software, DPI framework able to exploit multi-core system capabilities with the purpose of simplifying the development of network applications and to achieve the same level of performance of DPI tools based on proprietary silicon.

The remainder of the paper is structured as follows: Sec. 1 briefly outlines the DPI process. Sec. 2 introduces FastFlow, whereas Sec. 3 discusses the DPI framework overall design using FastFlow. Sec. 4 discusses the results of a set of experiments validating the design and implementation choices. Finally, Sec. 5 presents related work and Sec. 6 draws conclusions.

1. DPI

The identification of the application protocol (e.g. HTTP, SMTP, Skype, etc.) is performed by extracting the information contained in the packet header and often from the packet payload. To correctly extract the data carried by the protocol it is in general necessary to manage expensive operations so that this kind of processing is typically implemented (at least in part) through dedicated hardware. However, full software solutions are more appealing because they are more flexible and economical.

To identify the application flows, packets are classified in sets of packets all sharing the same <SOURCE IP ADDRESS, DESTINATION IP ADDRESS, SOURCE PORT, DESTINATION PORT, LAYER 4 PROTOCOL (E.G. TCP/UDP)> key. These sets are called "flows" and for each of them, in order to correctly reconstruct and identify the application communication, further information have to be stored in suitable data structures (typically an hash table). Considering the processing of TCP connections, when a TCP segment is received we need to perform the following steps:

Step1 Decode the packet headers in order to extract the 5-tuple key characterising the bidirectional flow to which the packet belongs.

Step2 Apply a symmetric hash function on the key, obtaining the bucket of the hash table containing the state of the flow of the received packet. We implemented some well known hash functions: Murmur3 [7], 32 bits FNV-1a [8], BKDR and a simple function computed by summing the 5 fields of the key. The programmer can use any of these functions when configuring the framework.

Step3 Access the table to obtain the current state of the flow.

Step4 Manage the TCP stream and, if the segment is out of order, store it for future reassembly by storing the packet payload.

Step5 Infer the protocol carried by the segment using specific *inspectors* (one for each supported protocol) by using previously collected information about the flow and analyzing the current packet. For example, to detect the HTTP protocol, the HTTP inspector could search inside the packet the strings representing HTTP methods (e.g. "GET", "PUT", "POST", etc.). To be more robust, it is also possible to use data collected from previously received packet for that flow, for example by correlating HTTP responses with HTTP requests.

Step6 If required, after protocol identification, further processing can be applied on the packet to extract protocol specific metadata information (e.g. HTTP URL, SMTP recipient, POP3 body, etc.).

If the flow protocol was already identified, the steps 4 and 5 can be skipped returning the result of the previous identification.

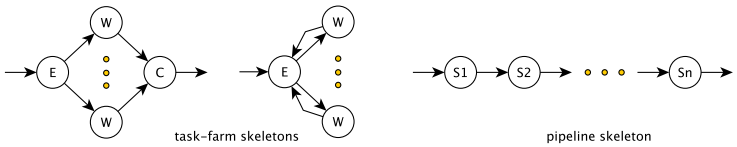

Figure 1. FastFlow skeletons: task-farm (with and without Collector) and pipeline.

2. FastFlow

FastFlow is a stream parallel programming framework providing the application programmers with ready-to-use, efficient and customizable stream parallel design skeletons [9]. FastFlow has been originally designed to target modern shared-cache multi-core. The result is a complete stream parallel programming framework able to successfully exploit parallelism in the computation of very fine grain tasks [10]. This is exactly what is needed to implement an high-performance parallel DPI engine, taking into account that single packet processing may require fairly small amount of work (even few tenths of clock cycles) *and*, the packet arrival rate may be extremely high.

FastFlow provides the application programmer with different, fully customizable and composable stream parallel skeletons including : a *pipeline* skeleton, with arbitrary number of stages, and a *task-farm* skeleton, with an arbitrary number of "worker" processes, each one independently executing tasks appearing on the input stream.

The farm skeleton is implemented using one or two additional concurrent entities, as shown in Fig. 1. One mandatory thread, the Emitter (E), which schedules tasks from the input stream towards the pool of worker threads. The default scheduling policy is round-robin, but different policies (e.g. *on-demand*, *broadcast*) are provided by the framework and, in addition, the application programmer may easily implement *ad-hoc* scheduling policies, if needed. Another (optional) thread, the Collector (C), eventually gathers the results computed by the worker threads and delivers them to the output stream. The default gathering policy is first-come first-served basis, but different policies are implemented in the framework and, if needed, may be used by the application programmer. If the Collector thread is not present, the task-farm skeleton may appear only as last stage of a FastFlow pipeline and the results are then delivered in main memory (or into files).

Communication channels between threads are implemented using lock-free Single-Producer Single-Consumer FIFO queues, with messages carrying data pointers rather than plain data copies [11].

3. Parallelisation using FastFlow

Considering that operations on different application flows are independent, the idea was to assign different groups of network flows to different concurrent modules such that they may be processed in parallel. Accordingly, we structured the framework (called PEAFOWL [12]) as a task-farm skeleton, having the flow table carefully partitioned among the set of workers. The requirement is that each worker processes only the network flows belonging to its own table partition. This way we avoid any true-sharing of the data-structure and the additional overhead of synchronization among workers.

In order to fulfill this requirement, we provided the FastFlow task-farm Emitter with an ad-hoc scheduling function which distributes to each worker exactly those packets

belonging to the flows contained in its partition. Consequently, the Emitter needs first to extract the key of the flow and then, as in the sequential case, to determine the bucket of the table where the flow has been allocated. Once the bucket has been found, the Emitter can easily derive the partition to which it belongs and, therefore, can forward the packet to the correct worker. Using this kind of parallelisation and considering the operations we described in Sec. 1, the Emitter sequentially performs steps 1. and 2. while each worker performs steps from 3. to 6. over different network flows in parallel. Apart from the

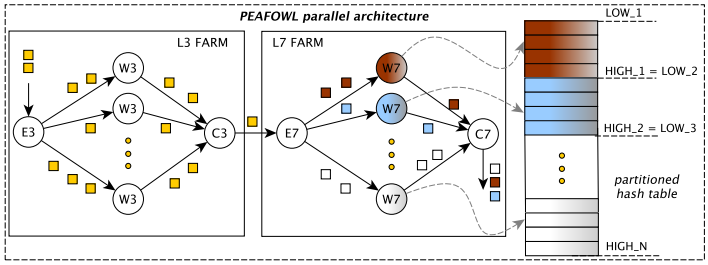

Figure 2. PEAFOWL structure when header parsing is parallelised using a task-farm (the L3 FARM).

communication channels between the Emitter and each worker, there is no data sharing, thus allowing each worker to advance in its execution without any further delay. Anyhow, false-sharing could happen between two threads accessing to some specific buckets (e.g. worker i accesses to the last bucket of partition i and worker $i + 1$ accesses the first bucket of partition $i + 1$). In this case we could have two different buckets used by two distinct threads on the same cache line producing data invalidation without having real sharing. Nevertheless, since the hash functions used produce well-distributed results (see [13]), the impact on performance of the false-sharing is very limited. However, since the latency required to parse the network and transport headers and to apply the hash function is not negligible, in some cases the Emitter may limit the framework scalability. To avoid this sequential bottleneck, in presence of high traffic rate, the Emitter is replaced with a second task-farm (L3 FARM) where each worker executes the header parsing steps (i.e. the same steps previously executed by the Emitter), obtaining a two-stage pipeline where each stage is a task-farm (L3 FARM and L7 FARM, respectively [1]), as shown in Fig. 2.

For the L3 FARM, any scheduling strategy can be used in principle. However, we want scheduling and gathering strategies suitable to preserve the ordering of the packets in such a way that they exit from the first stage in the same order they arrive from the network. In this way, if the packets belonging to the same flow were already ordered, they will arrive in the correct order to the second farm and immediatly after to the protocols inspectors. Consequently, the framework can avoid the overhead of TCP reordering when it is not really needed. In Sec. 4 we will analyze the performance improvement obtained by using an order preserving scheduling strategy for the L3 FARM.

4. Experimental results

In this section, the design choices as well as the overall performance of the framework is assessed using a set of experiments. The platform used for the experiments is a NUMA

[1] L3 stands for ISO/OSI Layers 3 and 4 processing. L7 stands for Layers 5 to 7 processing.

workstation having two INTEL XEON E5-2650 @ 2.00GHz nodes with a total of 16 cores (2-way hyperthreading). Each NUMA node has 16GB of main memory, 20MB of shared L3 cache, 256KB and 32KB of core private L2 and L1 caches, respectively.

To test the framework under the maximum load, we first measured the performance reading packets directly from the main memory of the platform (first loading the entire *pcap* file [14] at the beginning and then starting analyzing it), then we considered the performance when packets are read from a 10Gbits network card (NIC).

Dataset	*IPv4 packets*	*IPv4 flows*	*Description*
Synthetic	1428043	13314	Synthetic dataset
Darpa	1308081	38985	From `http://www.ll.mit.edu/mission/communications/cyber/CSTcorpora/ideval/data`
Local	524761	17939	Captured from local network

Table 1. Datasets used for the experiments

We used the datasets reported in Table 1. Each dataset is read multiple times in order to have sufficient input bandwidth. All experiments have been executed multiple times, and when the error bars are plotted, they represent any significant standard deviation from the mean value.

We start analyzing the number of packets per seconds successfully processed by the framework when only the protocol identification is executed on the 3 datasets considered. Figure 3 plots the bandwidth obtained varying the number of threads used. The framework has been set-up to use 2 task-farm (both L3 and L7), so we have 4 threads used for scheduling and gathering purposes (E3,C3 and E7,C7 in Fig. 2) and at least 1 worker thread for each of the two farms. As can be seen, the framework is able to process from 27.5 (*DARPA* dataset) to 36.3 (*Synthetic* dataset) Mpps (Millions of packets per second). Considering that a single sequential thread is able to sustain up to 4.2 Mpps we obtain a maximum speedup of $8.6\times$ for the protocol identification phase.

It is worth pointing out that, the protocol of an application flow is typically identified by inspecting only the first packets of the flow (typically the first 2-3 packets). For all the remaining packets, the cost of the inspection is not payed and each packet therefore incurs only in the hash-table access overhead (which is always needed for each packet to check if the flow was already identified). This is the reason why, on average, the processing of a single packet is a very fine grain operation. On the platform considered, the *Synthetic* dataset has an average computation time per packet of about 100ns.

We also analyzed the case where protocol identification alone is not sufficient for the application and further packet processing capabilities are required (*Step 6* in Sec. 1). We evaluated the scalability of the framework by increasing the computation time of the processing function executed for each identified packet. The scalability has been computed against the sequential time obtained when analyzing 500 times the *Synthetic* dataset and then varying the number of workers in the 2 farms in order to obtain the best performance. As sketched in Fig. 4, the obtained scalability is almost linear even for very low latency processing functions. When 12 worker threads are used in total (#W4+#W7), the configuration used is: $6 + 6$ for the protocol identification only, $4 + 8$ and $3 + 9$ for the cases 150ns and 250ns, respectively.

As described in Sec. 3, when the Emitter of the L7 FARM is parallelised and replaced by the L3 FARM, it is possible that packets leave the L3 FARM *out-of-order*. To

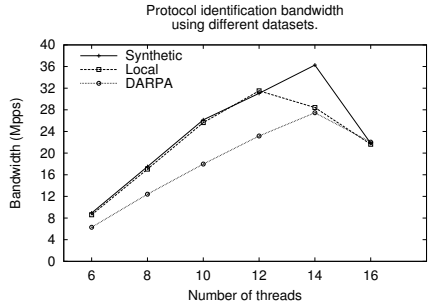

Figure 3. Performance of the protocol identification using different datasets

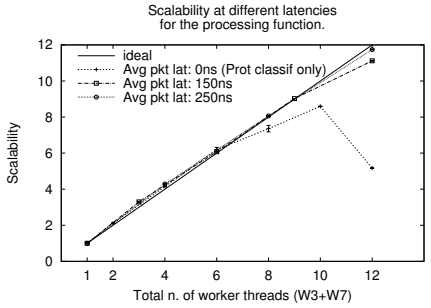

Figure 4. Scalability increasing packet processing computation time.

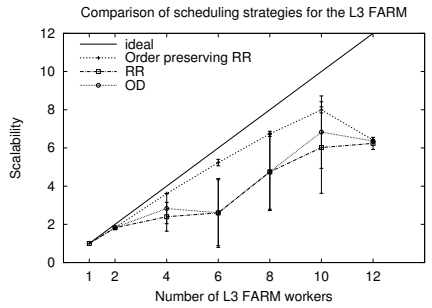

Figure 5. Comparison of different scheduling strategies for the L3 FARM.

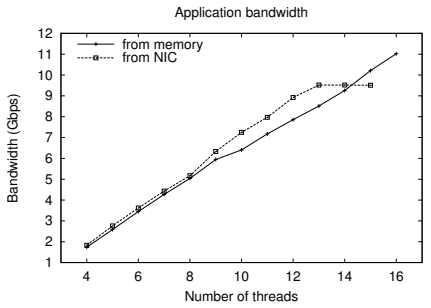

Figure 6. Bandwidth of the HTTP payload pattern matching application.

avoid this issue, the FastFlow task-farm skeleton has been extended with a scheduling strategy which preserves input/output packet ordering. Figure 5 compares the scalability obtained by the two non order preserving strategies, round-robin (RR) and on-demand (OD), which require an explicit reordering of packets in the Collector thread (C3 in Fig. 2), with the new strategy (Order preserving RR). The new strategy provides better results with respect to the other two both in terms of speedup and in terms of stability, because the overhead introduced by the ordered RR is smaller than the extra latency introduced by the explicit reordering of packets in the farm Collector. This can be explained by observing that the new strategy enforce ordering only when it is actually needed.

Finally, we measure the performance when the data is read from a 10Gbit INTEL 82598EB DUAL PORT NIC, with the two fiber ports connected back-to-back. The PF_RING Linux kernel module [15] has been used to improve the packet capture speed. The packets contained in the *Synthetic* dataset are sent over one network port using a packets sender application provided with the PF_RING kernel module, and they are read from the other port using the framework. Thanks to the low latency mechanisms provided by PF_RING module, we are able to read and write minimum packet sizes (60 bytes) at line rate from the NIC without any packet drop.

A simple demo application has been implemented to show the potential of the framework in a real setting. The application scans all the HTTP traffic searching for virus signatures. The check is performed by calling an HTTP callback in each worker thread for all packets containing a chunk of an HTTP body. The patter matching algorithm used is a

modified version of the Aho-Corasick pattern matching algorithm [16]. For this application scenario, the average computation granularity per packet is such that the Emitter of the L7 FARM is not a bottleneck of the system, thus the framework has been configured to use only one single task-farm. Figure 6 plots the bandwidth achieved varying the number of threads comparing the case when packets are read from main memory or directly from the NIC. The performance obtained in the latter case, is higher with regards to the one obtained reading packets from main memory. This can be explained considering that the card driver stores the first 64 bytes of the packet directly in the core cache of the Emitter thread (the one reading from the NIC), and this reduces both latency for the header parsing and memory contention.

Considering the experimental results, we can state that the PEAFOWL framework is able to obtain performance close to ideal for the HTTP virus signatures search application.

5. Related Work

With respect to well known existing open-source tools, instead of focusing on the number of supported protocols, we characterize this work by providing an efficient run-time support for multi-core architectures giving to the application programmer the possibility to specify in a simple way the callback used to process specific data carried by the protocol. In the Table 2 we compare PEAFOWL with OpenDPI/nDPI [3], libprotoident [5] and l7filter [4] against a number of important features.

	PEAFOWL	OpenDPI/nDPI	libprotoident	L7filter
IPv4 and IPv6 normalization	Yes	No	Yes	Yes
Flow management	Yes	No	Yes	Yes
TCP normalization	Yes	No	Yes	Yes
Arbitrary metadata processing	Yes	No	No	No
Multi-core support	Yes	No	No	No
Supported protocols	10	117/141	250	112

Table 2. Comparison between PEAFOWL with well known open-source DPI libraries.

From a performance perspective, we compare the sequential version of PEAFOWL with the nDPI library on the same platform used for the tests and under similar conditions (disabling for nDPI all not needed protocols), obtaining the results shown in Table 3.

	Bandwidth (Mpps)			% of Identified Traffic		
	Synthetic	Darpa	Local	Synthetic	Darpa	Local
nDPI	3.55	2.16	3.06	94.4%	29.3%	40.5%
PEAFOWL (seq.)	4.44	3.36	3.63	94.4%	29.7%	42.5%

Table 3. PEAFOWL (sequential version) vs. nDPI over different datasets

As can be seen in Table 3, PEAFOWL exhibits slightly better performance (and comparable quality for the protocols implemented) with respect to nDPI. However, since nDPI has been designed to support lots of application protocols, its internal data structures are bigger than the ones used in PEAFOWL (even though unused protocols have been disabled), therefore it has less advantages from cache spatial locality for the datasets considered.

6. Conclusions and Future Work

This paper presents the design and implementation of a framework called PEAFOWL for efficient DPI analysis on modern multi-core architectures.

The performance obtained shows that PEAFOWL is able to achieve good speedup also when the very low latency protocol identification tasks are executed. The results have been also validated studying the capabilities of the framework in a real HTTP packet inspection application where close to ideal performance is obtained. The measured performance clearly indicates that: i) the availability of a larger number of cores may further improve the bandwidth of our DPI framework thus paving the way to multi-10 Gbit traffic analysis, and ii) it is possible to successfully perform high-speed DPI analysis using commodity hardware and open software solutions.

As a future work we have planned to introduce adaptivity in the framework in order to automatically increase or decrease the number of concurrent threads in the FastFlow task-farm and to automatically modify the skeleton structure in order to avoid possible sequential bottleneck.

References

[1] YoungH. Cho, Shiva Navab, and WilliamH. Mangione-Smith. Specialized hardware for deep network packet filtering. In Manfred Glesner, Peter Zipf, and Michel Renovell, editors, *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, volume 2438 of *Lecture Notes in Computer Science*, pages 452–461. Springer Berlin Heidelberg, 2002.

[2] Jung-Sik Sung, Seok-Min Kang, Youngseok Lee, Taeck-Geun Kwon, and Bong-Tae Kim. A multi-gigabit rate deep packet inspection algorithm using tcam. In *Global Telecommunications Conference, 2005. GLOBECOM '05. IEEE*, volume 1, pages 5 pp.–, Dec.

[3] nDPI project website, 2013. `http://www.ntop.org/products/ndpi`.

[4] L7filter project website, 2013. `http://l7-filter.clearfoundation.com/`.

[5] Libprotoident project website, 2013. `http://research.wand.net.nz/software/libprotoident.php`.

[6] M. Becchi, M. Franklin, and P. Crowley. A workload for evaluating deep packet inspection architectures. In *Workload Characterization, (IISWC). IEEE International Symposium on*, pages 79–89, 2008.

[7] Murmur3 hash functions description, 2013. `http://code.google.com/p/smhasher/wiki/MurmurHash3`.

[8] FNV hash functions description, 2013. `http://www.isthe.com/chongo/tech/comp/fnv`.

[9] FastFlow project website, 2013. `http://mc-fastflow.sourceforge.net`.

[10] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. Accelerating code on multi-cores with fastflow. In E. Jeannot, R. Namyst, and J. Roman, editors, *Proc. of 17th Intl. Euro-Par 2011 Parallel Processing*, volume 6853 of *LNCS*, pages 170–181, Bordeaux, France, aug 2011. Springer.

[11] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. An efficient unbounded lock-free queue for multi-core systems. In *Proc. of 18th Intl. Euro-Par 2012 Parallel Processing*, volume 7484 of *LNCS*, pages 662–673, Rhodes Island, Greece, aug 2012. Springer.

[12] Peafowl project website, 2013. `https://github.com/DanieleDeSensi/Peafowl`.

[13] Daniele De Sensi. Dpi over commodity hardware: implementation of a scalable framework using fastflow. Master Degree in Computer Science and Networking, University of Pisa, Feb. 2013, Available at (in english): http://etd.adm.unipi.it/theses/available/etd-02042013-101033/.

[14] TCPDUMP&LibPcap project website, 2013. `http://www.tcpdump.org/`.

[15] Luca Deri. Improving passive packet capture:beyond device polling. In *Proceedings of the 4th International System Adinistration and Network Engineering Conference (SANE 2004)*, Sep 2004.

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