Network Monitoring on Multicores with Algorithmic Skeletons

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

Abstract. Monitoring network traffic on 10 Gbit networks requires very efficient tools capable of exploiting modern multicore computing architectures. Specialized network cards can accelerate packet capture and thus reduce the processing overhead, but they can not achieve adequate packet analysis performance. For this reason most monitoring tools cannot cope with high network speeds.

We describe the design and implementation of ffProbe, a network traffic monitoring application built on top of FastFlow, combined with several optimized parallel programming patterns. We compare ffProbe with two popular network monitoring probes. The results demonstrate that it can scale significantly better with number of cores and thus may be suitable for monitoring 10 Gbit networks using commodity servers.

Keywords. Network traffic monitoring, algorithmic skeletons, pipeline, structured parallel programming, NetFlow, multi core, FastFlow.

1. Introduction

The increasing number of applications running on top of IP networks is making ever more urgent the need to implement very efficient network monitoring tools that can be used for analysis of network traffic in differing contexts. Most network device vendors provide proprietary hardware and software tools to support network monitoring and suitable protocols have been developed to allow data monitored by different hardware to be analyzed [3]. In most cases, high performance network monitoring is implemented on dedicated hardware: special network cards and/or special processors [13,9].

With the recent shift from single core to multicore processing elements, more and more computing power is available on a single socket and nowadays architectures may be bought for a few thousand dollars that sport up to 48 general purpose cores. The resulting computing power makes feasible the possibility of implementing high speed/high traffic network inspection/monitoring tools on non-dedicated hardware, rather than only on specialized hardware. However, efficient multicore parallel program development, such as that needed to implement this sort of network monitoring application, usually requires substantial programming effort. While multicore programming environments such as OpenMP provide the programmer with simple and quite intuitive programming abstractions, the fine tuning of parallel applications on multicores is still a time consuming activity, requiring extensive knowledge of the multicore features when ultimate performances are looked for.

To avoid such problems, advanced *structured parallel programming environments* have been developed. These programming environments raise the level of abstraction presented to the application programmer as compared to that presented by more classical environments such as OpenMP. Rather than exposing mechanisms that the application programmer may freely combine to implement his/her parallel application, these environments provide the application programmer with a set of *parallel patterns* that can be used–alone or in composition–as building blocks for implementing a wide range of parallel applications. In the work described here, we used one of these structured programming frameworks to implement an efficient network monitoring application. The framework used is FastFlow, providing the application programmer with a set of very efficient parallel patterns including pipelines and task farms. These patterns are build on top of a compact layer implementing lock-free and fence-free single producer single consumer channels [2]. The resulting framework demonstrates the feasibility of efficient high-speed network monitoring on commodity network servers and Ethernet adapter hardware.

The remainder of the paper is organized as follows: Sec. 2 and Sec. 2.1 introduce high-speed traffic monitoring and related work. Sec. 3 discusses our original implementation of the NetFlow probe on top of FastFlow. Sec. 4 discusses the experiments and the results validating the proposed probe architecture and finally Sec. 5 draws conclusions.

2. High-Speed Traffic Monitoring

The shift from 1 Gbit to 10 Gbit networks has pushed hardware manufacturers to find new solutions for exploiting multicore architectures. The first goal is to use all the available cores for improving packet receive/transmission. For this reason, modern network adapters feature multi-queue RX/TX, so that a physical network adapter is logically partitioned into several logical adapters each sharing the same MAC address and Ethernet port. Incoming packets are decoded in hardware, and a hash value based on various packet fields such as IP address, protocol and port is computed. Based on the hash value, the network adapter places the packets in a specific queue. This way the kernel can simultaneously poll and transmit packets from each queue, thus maximizing the overall performance. Unfortunately the operating systems are not mature enough to exploit this feature, as they do not expose queues to user-space applications thus limiting them to viewing the network adapter as a single entity. The outcome is that fetching packets in parallel from the network adapter is not possible unless applications can directly access the various queues. PF_RING [8] is a packet processing framework that we have developed that implements various mechanisms for enhancing packet processing and that also allows applications to natively access adapters' queues. Recently PF_RING has been enhanced with support of 10 Gbit DNA (Direct NIC Access) that allows applications to receive/transmit packets while completely bypassing the kernel, as they can access directly the queues that have been previously mapped into user space memory. This means that the cost of receiving/transmitting a packet is basically the cost of a memory read/write, thus making 10 Gbit wire-rate packet RX/TX now manageable using commodity network adapters.

Efficient packet receive/transmit is not the only problem in traffic monitoring. NetFlow [3] is the industry standard protocol for traffic monitoring. In NetFlow, packets are classified in flows. A flow is defined as a set of packets all sharing common properties such as source/destination IP address/port, protocol, ingress interface and type of service. NetFlow probes are the applications responsible for managing flows. Often they are implemented in routers, but they can also be implemented as software applications running on servers. Flows are created when the first packet of a given flow has been received, and they expire when a specified amount of time (e.g. 2 mins) has elapsed or if no packets for the flow have been received for fixed period. When a flow has expired, it is packed with other flows into a UDP packet formatted according to the NetFlow format and sent to a NetFlow collector application that receives and stores it.

2.1. Related Work

Traditionally Netflow probes are embedded into network devices such as routers and L3 switches [1]. However the cost of specialized cards for implementing Netflow on routers has prompted developers to create software probes that can run on standard servers. A further driving force for software probes is the ability to create specialized probes [10] focusing on selected traffic (e.g. http, peer-to-peer, and voice-over-IP) that would be too costly to implement on network devices. On the other hand, the use of accelerating card has ignited the development of hardware-accelerated Netflow probes [15]. Pure software probes have been accelerated by moving some computationally expensive functions into the kernel [7], [12], [4] to achieve performance close to line-rate using commodity hardware. *nProbe* [5] is a mature NetFlow probe supporting various flow formats and available under the GPLv2 license. It can be used both as a single and multi-threaded application. In single thread mode, incoming packets are captured, decoded, and processed by the same thread. In multi-thread mode, there is one thread that captures packets and n processing threads, with the communication among threads being implemented by means of lock-free queues where incoming packets are copied after header hashing. Although nProbe can sit on top of the widespread pcap API, to optimize performance, it exploits some unique DNA features such as zero-copy packet capture and hardware-based flow hashing to avoid computing in software flow hashes. This latter feature allows nProbe to be started in standard or quick mode. In quick mode, nProbe uses the hardware-computed flow hash to search for the flow using the hash, for increased performance. *Softflowd* is an open-source software NetFlow traffic probe [11]. It supports various NetFlow versions and both IPv4 and IPv6 flows. Softflowd is single threaded: it cannot be configured to spawn multiple threads to increase processing speed. Internally flows are kept on a red-black tree. Whenever a packet is received, Softflowd decodes it and updates the tree. Flows to be exported are computed by periodically navigating the tree according to various criteria such as maximum flow lifetime and idle flow time. From the software point of view, Softflowd is a fairly simple application whose efficiency comes from its very light internal design.

3. Probe Architecture

In this Section we describe the architecture of ffProbe in comparison with two other popular open-source traffic probes: *nProbe* (see `http://www.ntop.org/nProbe.html`) and *softflowd* (see `http://code.google.com/p/softflowd/`).

ffProbe is a new pipelined parallel implementation of a NetFlow [3] probe built on top of FastFlow, a parallel programming framework for multicore platforms based on non-blocking lock-free/fence-free synchronization mechanisms [6,2]. The FastFlow framework is composed of a stack of layers that progressively abstracts out the programming of shared-memory parallel applications. The goal of the stack is twofold: to ease the development of applications and to make them very fast and scalable. FastFlow is particularly targeted to the development of streaming applications.

FastFlow proved very effective for implementing and experimenting with different versions of ffProbe, using different parallel patterns. The base ffProbe pipeline uses two different types of stage: *reader* and *worker* (see Fig. 1, A). The reader stage captures packets from the device using `PF_RING` [8] and for each packet extracts the information needed by the probe (Source IP, L4 Source port, etc...). A hash function is applied to this information and based on the returned value the reader decides to which worker this packet is directed. The "task" that the reader sends forward to the workers list contains a number of packets that are directed to different workers in the pipeline. Each worker will fetch and process only its own packets. Eventually, the reader sends the task to the first worker of the pipeline using the FastFlow SPSC (*Single Producer Single Consumer*) bounded queue.

Extra readers can be used to try to improve performance. In this case each reader will read from a different network interface or from different queues of the same interface. The communication between the m readers and the first worker is implemented with m independent SPSC FastFlow lock-free queues. The worker will extract the packets from the input queues according to a non-blocking round-robin policy.

The ffProbe pipeline is parametric in the number of workers and this allows increase or reduction of the pipeline length according to the resources available and to the inter departure time of the reader. Each worker has a private hash table and the union of all these tables represents the (logically global) table where all the flows are stored. By avoiding any sharing of the table among worker stages[1] we avoid the typical problems related to the cache invalidation.

Each worker, extracts from the received task the packets contained in its list and updates its own local hash table partition with the data relative to these packets. To reduce the latency of this phase, while a collision list is scanned to find the flow to update, the collision list of the next flow is prefetched. After the update the worker checks a number of flows in order to see if any of them has expired. The expired flows are then inserted into another list of the task and this is sent to the next worker.

When the task eventually reaches the last worker, after the classical operations, the exporting routine is called. This routine stores the expired flows and, if there is a sufficient number of flows to be exported, sends them to the remote NetFlow collector using UDP. If the latency of this routine is seen to be too high, it is possible to execute it in an independent stage of the pipeline, located directly after the workers.

ffProbe maps the threads of the pipeline in such a way that over provisioning and inter-chip communications are avoided. The user may manually modify this mapping, if needed.

Using FastFlow structured programming model, we implemented several versions of ffProbe, differing slightly in the kind of parallel pattern used. In particular, we used the base ffProbe pipeline to build different configurations:

P1 A single pipeline composed by one reader and n workers (Fig. 1, A).

[1] Each worker accesses a *portion* of the table.

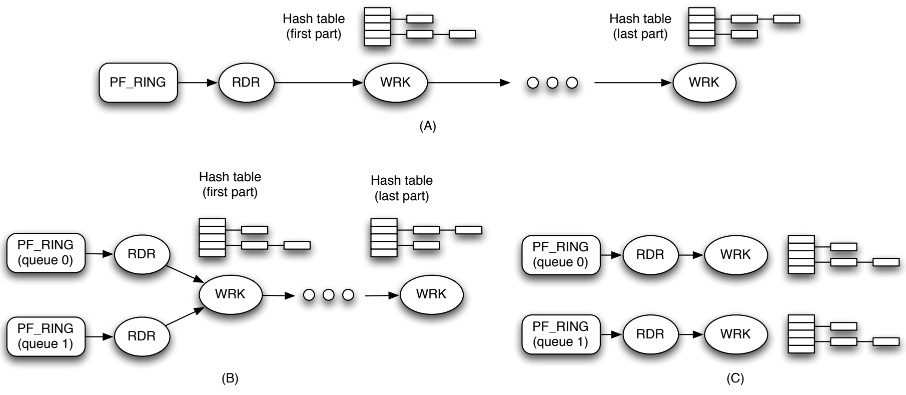


Figure 1. Different possible structuring of ffProbe.

P2 A single pipeline with m readers and n workers (Fig. 1, B).
P3 m pipelines, one for each queue (or interface). Each pipeline has one reader and n workers (Fig. 1, C, with $n = 1$)).

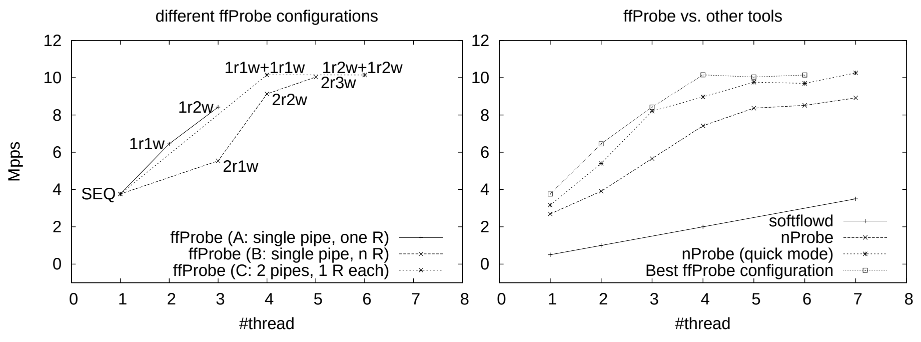
All these different parallel versions of ffProbe have been experimented with and the associated results are presented and discussed in Sec. 4. It is worth pointing out that implementing the different versions required modest programming effort, due to the facilities offered by FastFlow in terms of parallel patterns and pattern composition. Similar parallel patterns may be used to implement distributed versions of ffProbe, with one node capturing packets and directing them to ffProbe pipelines running on different nodes, identified through appropriate hashing of the packet header.

4. Validation

In order to evaluate the ffProbe performance, we have used two different servers, namely a dual quad core Xeon E5520 Nehalem (directory based cache coherency, with hyper threading support, 8Mbytes Smart cache, running at 2.26GHz), and a single CPU 8-core server based on the Intel Sandy Bridge controller and on low-end Xeon E3-1230 running at 3.20 GHz. Both servers were equipped with a dual port 10 Gbit Intel-based Silicom NICs. Ethernet ports were connected back-to-back with a crossed optical fiber patch, so that one port is used for generating traffic and the other for receiving it. To minimize the time spent capturing packets, we used PF_RING DNA (Direct NIC Access) that allows the kernel to be bypassed when both sending and receiving packets. Applications access packets by reading them in zero-copy mode from the NIC's memory though the PF_RING API. Traffic has been generated by replying at wire rate with various traffic samples using `pfsend`, an application that is part of the PF_RING framework. The traffic samples used include both synthetic and real traffic captured on a network. In order to evaluate the application performance in the worst case, we have also created a few traffic samples that contain up to one million concurrent flows, all starting and ending at the same time. Such traces put significant pressure on applications as the analyzed flows are both created and expired at the same time resulting in a peak of workload. By contrast, traces captured from real networks, although containing traffic spikes, do not represent the worst case, as due to variable size packets, the greatest observed rate does not exceed 5Mpps, which is much less of what we can generate with a synthetic trace. Such traces contain minimal size packets (60 bytes) so that the network receive rate can match the maximum theoretical rate. Note that real networks on average are not loaded above 50% of their capacity as network operators want to avoid dropping packets due to traffic peaks. Furthermore the average packet size on real networks is 512 bytes which translates to 2.35 Mpps. As `pfsend` and probes both run on the same physical machine, we have reserved one core to `pfsend`, and so probes can use up to 7 processor cores. Each RX queue has been mapped to the corresponding core, so that $i-th$ core is mapped to the $i-th$ RX queue. During tests we configured probes to export flows every minute using the NetFlow v5 format that is the common flow format supported by all probes.

Evaluating ffProbe We ran several tests using ffProbe. In particular we looked for top performances and scalability when using different ffProbe structures, such as those mentioned in Sec. 3. Fig. 2 shows the results obtained from analyzing traffic containing 500K synthetic flows as well as the memory footprints measured in the different experiments. As different versions of the ffProbe application are only significant for particular reader and worker stages configurations in the ffProbe pipeline(s), the figures report only results for the meaningful configurations, and thus values relative to some parallelism degrees are missing. In particular, in Fig. 2 (right) the ffProbe curve stops one step before those of the other tools as in this case we used one core to generate the traffic. A single pipeline (such as the one in Fig. 1, configuration A) performs quite well, scaling with the increased number of threads used. It is worth pointing out that the configuration “1R + 1W” doubles the performance of the sequential probe, as the split of work between reader and worker in this case already guarantees a notable performance increment. The configuration B in Fig. 1 was not as good as the previous one, indeed. The problem here is that with two readers we are capturing too much traffic to be managed by only one worker. However, increasing the number of workers we get performances comparable with the other solutions. By using two pipelines (Fig. 1, configuration C), each reading from its own `PF_RING` queue, ffProbe reached the same performance as nProbe, while using only 4 threads. With this solution the pipelines are executed in independent processes so we can avoid inter-chip communication. This allows us to increase the parallelism degree and to manage higher rates, provided that the threads belonging to the same process are executed on cores on the same chip. It’s important to point out that with respect to the other solutions we succeeded achieving good performance figures even using a single multi-threaded process. This is important because in case of bidirectional flows we should carefully avoid to use more processes due to the direction dependent behaviour of the hash function computed by the network card. We also performed some tests with one million flows, and we obtained a peak performance of 9.68 Mpps (without losses) with 6 threads. Overall, the results achieved using ffProbe are significantly better than those achieved using multicores with more conventional programming techniques (e.g. [14]). In all cases, the CPU utilization for the cores used was close to 100%, which is in part due to the peculiarities of the FastFlow implementation: in order to guarantee lock and fence free communications, the fine grain contention is solved using busy waiting.

Comparing with nProbe and softflowd As nProbe has been optimized for single-thread mode, we have tested it spawning one instance per RX queue. A set of single threaded



Program version	P1			P2			P3	
#Thread	1	2	3	3	4	5	4	6
Memory footprint	50M	94M	78M	115M	150M	150M	64M	64M

Figure 2. Performance and memory footprint comparison: different versions of ffProbe, nProbe, softflowd.

nProbe instances in standard mode (each processing a different message queue) can handle up 8.9 Mpps, whereas in quick mode 10.25 Mpps (see Fig. 2 right). The use of threads can improve the processing speed of the single nProbe instance, even though the scalability is not linear: adding more than two threads is of little/no help. We believe that, although nProbe includes lock-free data structures, when using multiple threads incoming packets need to be copied into a queue, and so we lose the zero-copy advantage of DNA. Furthermore due to the application design, increasing the number of processing threads leaves fewer cycles to the export thread, thus jeopardizing the performance. For this reason we believe that the optimal solution is to use one single threaded nProbe instance per queue. The memory footprint of nProbe is slightly larger that that of ffProbe, mainly due to the larger buckets used to store information, while the CPU utilization, as in ffProbe, is close to 100%. As `softflowd` is a single threaded application, we can test it in only one configuration and spawn one instance per core and RX queue. On average we can handle about 500K pps per instance which translates to about 3.5 Mpps in total when using 7 cores. Last but not least, in the right graph of Fig. 2, ffProbe numbers are relative to execution on a processor slower than the one used for the nProbe and softflowd executions, due to the different periods of availability of the two machines equipped with the identical 10Gbit cards. We therefore expect ffProbe results are even better compared with nProbe.

In our opinion these results show that ffProbe exploits limited degree multicore architectures better that other NetFlow probe implementations and the scalability demonstrated indicates better results may be achieved on larger multicore configurations.

5. Conclusions

This paper has confirmed that modern, pattern based parallel programming techniques can be profitably used to implement a NetFlow probe on state-of-the-art multicore architectures. The pattern based programming environment allowed experimentation with different versions (as far as parallel exploitation is concerned) of the probe with relatively little programming effort. The validation has confirmed that different parallel structuring of the probe achieve different peak performances, and that some of them scale quite well with the number of cores. The processing performance observed during experiments confirmed that this technology is able to monitor multi-10 Gbit networks when using average size packets, making it suitable for large scale monitoring deployments.

Acknowledgements We would like to thank Alfredo Cardigliano <`cardigliano@ntop.org`> for his support during the validation, Massimo Torquati <`torquati@di.unipi.it`> for his support as FastFlow developer and Silicom Inc. for providing the 10 Gbit cards used in our tests.
This work has been partially supported by EU FP7 STREP ParaPhrase.

References

[1] F. Afanasiev, A. Petrov, V. Grachev, and A. M. Sukhov. Flow-based analysis of internet traffic. *CoRR*, cs.NI/0306037, 2003.

[2] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. Accelerating code on multicores 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.

[3] B. Claise. Cisco Systems NetFlow Services Export Version 9 - RFC 3954, 2004. `http://www.faqs.org/rfcs/rfc3954.html`.

[4] E. L. Congduc, E. Lemoine, C. Pham, and L. Lefevre. Packet classification in the NIC for improved SMP-based Internet servers. In *IEEE Proceedings of the International Conference on Networking (ICN 2004*, 2004.

[5] L. Deri. nProbe: an Open Source NetFlow probe for Gbit Networks. In *Proc. of Terena TNC*, 2003.

[6] FastFlow home page, 2011. `http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about`.

[7] A. Ferro, F. Liberal, A. Muñoz, and C. Perfecto. Network traffic sensor for multiprocessor architectures: Design improvement proposals. In P. Dini, P. Lorenz, and J. de Souza, editors, *Service Assurance with Partial and Intermittent Resources*, volume 3126 of *Lecture Notes in Computer Science*, pages 146–157. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-27767-5.15.

[8] F. Fusco and L. Deri. High speed network traffic analysis with commodity multi-core systems. In *Proceedings of the 10th annual conference on Internet measurement*, IMC '10, pages 218–224, New York, NY, USA, 2010. ACM.

[9] J. Novotny, P. Celeda, T. Dedek, and R. Krejcy. Hardware Acceleration for Cyber Security. In *Proceedings of the IST-091 - Information Assurance and Cyber Defence*, 2010. Tallinn, Estonia : NATO Research and Technology Organization, 2010.

[10] D. Rossi and S. Valenti. Fine-grained traffic classification with netflow data. In *Proceedings of the 6th International Wireless Communications and Mobile Computing Conference*, IWCMC '10, pages 479–483, New York, NY, USA, 2010. ACM.

[11] `softflowd` home page, 2011. `http://www.mindrot.org/projects/softflowd/`.

[12] P. Wang and Z. Liu. Operating System Support for High-Performance Networking, A Survey. *Sensors*, 11(6), 2011. available at `http://www.mdpi.com/1424-8220/11/6/5900/`.

[13] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27:15–31, September 2007.

[14] C. Xu, W. Shi, and Q. Xiong. An Architecture for Parallelizing Network Monitoring Based on Multi-Core Processors. *Journal of Convergence Information Technology*, 6(4), April 2011.

[15] M. Zadnik and L. Lhotka. Hardware-accelerated netflow probe. Technical Report TR-32-2005, CESNET, Czec Republic, Dec. 2005.