High Speed Network Traffic Analysis with Commodity Multi-core Systems

Francesco Fusco
IBM Research - Zurich
ETH Zurich
ffu@zurich.ibm.com

Luca Deri
ntop
deri@ntop.org

ABSTRACT

Multi-core systems are the current dominant trend in computer processors. However, kernel network layers often do not fully exploit multi-core architectures. This is due to issues such as legacy code, resource competition of the RX-queues in network interfaces, as well as unnecessary memory copies between the OS layers. The result is that packet capture, the core operation in every network monitoring application, may even experience performance penalties when adapted to multi-core architectures. This work presents common pitfalls of network monitoring applications when used with multi-core systems, and presents solutions to these issues. We describe the design and implementation of a novel multi-core aware packet capture kernel module that enabled monitoring applications to scale with the number of cores. We showcase that we can achieve high packet capture performance on modern commodity hardware.

Categories and Subject Descriptors

D.4.4 [**Operating Systems**]: Communications Management; C.2.3 [**Network Operations**]: Network monitoring

General Terms

Measurement, Performance

Keywords

Linux kernel, network packet capture, multi-core systems

1. INTRODUCTION

The heterogeneity of Internet-based services and advances in interconnection technologies raised the demand for advanced passive monitoring applications. In particular analyzing high-speed networks by means of software applications running on commodity off-the-shelf hardware presents major performance challenges.

Researchers have demonstrated that packet capture, the cornerstone of the majority of passive monitoring applications, can be substantially improved by enhancing general purpose operating systems for traffic analysis [10, 11, 25]. These results are encouraging because today's commodity hardware offers features and performance that just few years ago were only provided by costly custom hardware design. Modern network interface cards offer multiple TX/RX queues and advanced hardware mechanisms able to balance traffic across queues. Desktop-class machines are becoming advanced multi-core and even multi-processor parallel architectures capable to execute multiple threads at the same time.

Unfortunately, packet capture technologies do not properly exploit this increased parallelism and, as we show in our experiments, packet capture performance may be reduced when monitoring applications instantiate several packet capture threads or multi-queues adapters are used. This is due to three major reasons: a) resource competition of threads on the network interfaces RX queues, b) unnecessary packet copies, and c) improper scheduling and interrupt balancing.

In this work, we mitigate the above issues by introducing a novel packet capture technology designed for exploiting the parallelism offered by modern architectures and network interface cards, and we evaluate its performance using hardware traffic generators. The evaluation shows that thanks to our technology a commodity server can process more than *4 Gbps* per physical processor, which is more than *four* times higher than what we can achieve on the same hardware with previous generation packet capture technologies.

Our work makes several important contributions:

- We successfully exploit traffic balancing features offered by modern network adapters and make each RX queue visible to the monitoring applications by means of *virtual capture devices*. To the best of our knowledge, this work describes the first packet capture technology specifically tailored for modern multi-queue adapters.
- We propose a solution that substantially simplifies the development of highly scalable multi-threaded traffic analysis applications and we released it under an open-source license. Since compatibility with the popular libpcap [4] library is mostly preserved, we believe that it can smooth the transition toward efficient parallel packet processing.
- We minimize the memory bandwidth footprint by reducing the per-packet cost to a single packet copy, and

optimize the cache hierarchy utilization by combining lock-less buffers together with optimal scheduling settings.

2. MOTIVATION AND SCOPE OF WORK

Modern multi-core-aware network adapters are logically partitioned into several RX/TX queues where packets are flow-balanced across queues using hardware-based facilities such as RSS (Receive-side Scaling) part of Intel^{TM}I/O Acceleration Technology (I/O AT) [16, 17]. By splitting a single RX queue into several smaller queues, the load, both in terms of packets and interrupts, can be balanced across cores to improve the overall performance. Modern interface cards (NICs) support static or even dynamically configurable [12] balancing policies. The number of available queues depends on the NIC chipset, and it is limited by the number of available system cores.[1]

However, in most operating systems, packets are fetched using packet polling [22, 24] techniques that have been designed in the pre-multi-core age, when network adapters were equipped with a single RX queue. From the operating system point of view, there is no difference between a legacy 100 Mbit card and a modern 10 Gbit card as the driver hides all card, media and network speed details. As shown in Figure 1 device drivers must merge all queues into one as it used to happen with legacy adapters featuring a single queue. This design limitation is the cause of a major performance bottleneck, because even if a user space application spawns several threads to consume packets, they all have to compete for receiving packets from the same socket. Competition is costly as semaphores or similar techniques have to be used in order to serialize this work instead of carrying it out in parallel.

Even if multi-core architectures, such as the one depicted in Figure 2, are equipped with cache levels dynamically shared among different cores within a CPU, integrated memory controllers and multi-channel memories, memory bandwidth has been identified as a limiting factor for the scalability of current and future multi-core processors [6, 23]. In fact, technology projections suggest that off-chip memory bandwidth is going to increase slowly compared to the desired growth in the number of cores. The *memory wall* problem represents a serious issue for memory intensive applications such as traffic analysis software tailored for high-speed networks. Reducing the memory bandwidth by minimizing the number of packet copies is a key requirement to exploit parallel architectures.

To reduce the number of packet copies, most capture packet technologies [10] use memory mapping based zero-copy techniques (instead of standard system calls) to carry packets from the kernel level to the user space. The packet journey inside the kernel starts at the NIC driver layer, where incoming packets are copied into a temporary memory area, the socket buffer [7, 20], that holds the packet until it gets processed by the networking stack. In network monitoring, since packets are often received on dedicated adapters not used for routing or management, socket buffer's allocations and deallocations are unnecessary and zero-copy could start directly at the driver layer and not just at the networking layer.

[1]For example on a quad-core machine we can have up to four queues per port.

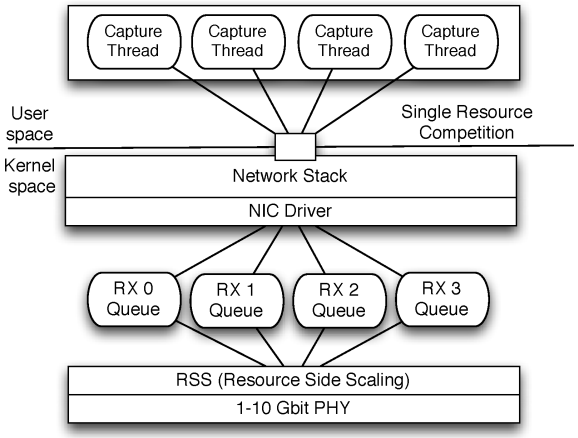

Figure 1: Design limitation in Network Monitoring Architectures.

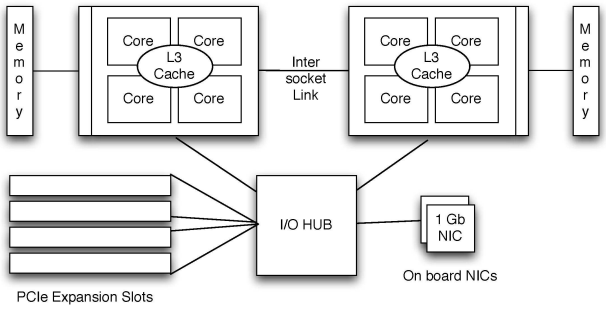

Figure 2: Commodity parallel architecture.

Memory bandwidth can be wasted when cache hierarchies are poorly exploited. Improperly balancing interrupt requests (IRQs) may lead to the excessive cache misses phenomena usually referred as cache-trashing. In order to avoid this problem, the interrupt request handler and the capture thread that consumes such packet must be executed on the same processor (to share the L3 level cache) or on the same core with Hyper-Threaded processors. Unfortunately, most operating systems uniformly balance interrupt requests across cores and schedule threads without considering architectural differences between cores. This is, in practice, a common case of packet losses. Modern operating systems allow users to tune IRQ balancing strategy and override the scheduling policy by means of CPU affinity manipulation [19]. Unfortunately, since current operating systems do not deliver queue identifiers up to the user space, applications do not have enough information to properly set the CPU affinity.

In summary, we identified two main issues that prevent parallelism to be exploited:

- There is a single resource competition by multi-threaded applications willing to concurrently consume packets coming from the same socket. This prevents multi-queue adapters to be fully exploited.

- Unnecessary packet copies, improper scheduling and interrupt balancing cause a sub-optimal memory bandwidth utilization.

The following section describes a packet capture architecture that addresses the identified limitations.

3. TOWARDS MULTI-CORE MONITORING ARCHITECTURES

We designed a high performance packet capture technology able to exploit multi-queue adapters and modern multi-core processors. We achieve our high performance by introducing virtual capture devices, with multi-threaded polling and zero-copy mechanisms. Linux is used as the target operating system, as it represents the de-facto reference platform for the evaluation of novel packet capture technologies. However, the exploited concepts are general and can also be adapted to other operating systems.

Our technology natively supports multi-queues and expose them to the users as *virtual capture devices* (see Figure 3). Virtual packet capture devices allow applications to be easily split into several independent threads of execution, each receiving and analyzing a portion of the traffic. In fact monitoring applications can either bind to a physical device (e.g., eth1) for receiving packets from all RX queues, or to a virtual device (e.g., eth1@2) for consuming packets from a specific queue only. The RSS hardware facility is responsible to balance the traffic across RX queues, with no CPU intervention.

The concept of virtual capture device has been implemented in PF_RING [10], a kernel level network layer designed for improving Linux packet capture performance. It also provides an extensible mechanism for analyzing packets at the kernel-level. PF_RING provides a zero-copy mechanism based on memory mapping to transfer packets from the kernel space to the user space without using expensive system calls (such as *read()*). However, since it sits on top of the standard network interface card drivers, it is affected by the same problems identified in the previous section. In particular, for each incoming packet a temporary memory area, called socket buffer [7, 20], is allocated by the network driver, and then copied to the PF_RING ring buffer which is memory mapped to user space.

TNAPI drivers: For avoiding the aforementioned issue, PF_RING features a zero-copy ring buffer for each RX queue and it supports a new NIC driver model optimized for packet capture applications called TNAPI (Threaded NAPI[2]).

TNAPI drivers when used with PF_RING completely avoid socket buffer's allocations. In particular, packets are copied directly from the RX queue to the associated PF_RING ring buffer for user space delivery. This process does not require any memory allocation because both the RX queue and the corresponding PF_RING ring are allocated statically. Moreover, since PF_RING ring buffers are memory-mapped to the user-space, moving packets from the RX queue ring to the user space requires a *single* packet copy. In this way, the driver does not deliver packets to the legacy networking stack so that *the kernel overhead is completely avoided*. If desired, users can configure the driver to push packets into the standard networking stack as well, but this configuration is not recommended as it is the cause of a substantial performance drop as packets have to cross legacy networking stack layers.

[2]NAPI is the driver model that introduced polling in the Linux kernel

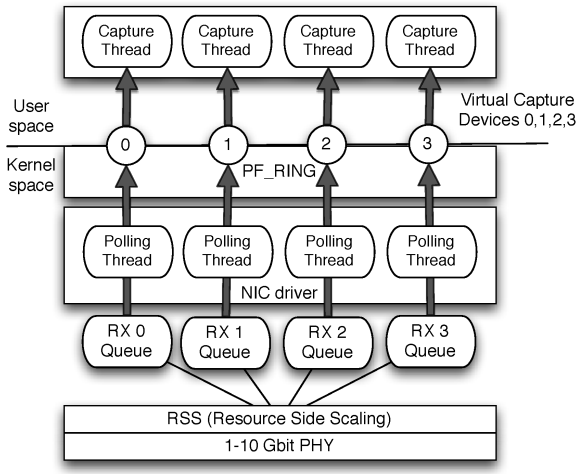

Figure 3: Multi-queue aware packet capture design. Each capture thread fetches packets from a single Virtual Capture Device.

Instead of relying on the standard kernel polling mechanisms to fetch packets from each queue, TNAPI features in-driver *multi-threaded packet polling*. TNAPI drivers spawn one polling thread for for each RX queue (see Figure 3). Each polling thread fetches incoming packet(s) from the corresponding RX queue, and, passes the packet to PF_RING. Inside PF_RING, packet processing involves packet parsing and, depending on the configuration, may include packet filtering using the popular BPF filters [4] or even more complex application level filtering mechanisms [15]. Kernel level packet processing is performed by polling threads in *parallel*.

TNAPI drivers spawn polling threads and bind them to a specific core by means of CPU affinity manipulation. In this way the entire traffic coming from a single RX queue is always handled by the same core at the kernel level. The obvious advantage is the increased cache locality for poller threads. However, there is another big gain that depends on interrupt mitigation. Modern network cards and their respective drivers do not raise an interrupt for every packet under high-rate traffic conditions. Instead, drivers disable interrupts and switch to polling mode in such situations. If the traffic is not properly balanced across multi-queues, or if simply the traffic is bursty, we can expect to have busy queues working in polling mode and queues generating interrupts. By binding the polling threads to the same core where interrupts for this queue are received we prevent threads polling busy queues to be interrupted by other queues processing low-rate incoming traffic.

The architecture depicted in Figure 3 and implemented in TNAPI, solves the single resource competition problem identified in the previous section. In fact, users can instantiate one packet consumer thread at the user space level for each virtual packet capture device (RX queue). Having a single packet consumer per virtual packet capture device does not require any locking primitive such as semaphores that, as a side effect, invalidate processor caches. In fact, for each RX queue the polling thread at the kernel level and the packet consumer thread at the user space level exchanges packets through a lock-less Single Reader Single Writer (SRSW) buffer.

In order to avoid cache invalidation due to improper

scheduling, users can manipulate the CPU affinity to make sure that both threads are executed on cores or Hyper-Threads sharing levels of caches. In this way, multi-core architectures can be fully exploited by leveraging high bandwidth and low-latency inter-core communications. We decided not to impose specific affinity settings for the consumer threads, meaning that the user level packet capture library does not set affinity. Users are responsible to perform fine grained tuning of the CPU affinity depending on how CPU intensive is the traffic analysis task. This is straightforward and under Linux requires a single function call.[3] It is worth noting, that fine-grained tuning of the system is simply not feasible if queue information is not exported up to the user space.

Compatibility and Development Issues: For compatibility reasons we ported the popular libpcap [4] library on top of our packet capture technology. This makes the porting of already existing monitoring applications over the new packet capture technology trivial and simplifies the development of new ones. As of today, we implemented packet capture optimized drivers for popular multi-queue Intel^{TM}1 and 10 Gbit adapters (82575/6 and 82598/9 chips).

4. EVALUATION

We evaluated the work using two different parallel architectures belonging to different market segments (low-end and high-end) equipped with the same Intel multi-queue network card. Details of the platforms are listed in Table 1. An IXIA 400 [3] traffic generator has been used to inject the network traffic for experiments. For 10 Gbit traffic generation, several IXIA-generated 1 Gbit streams have been merged into a 10 Gbit link using a HP ProCurve switch. In order to exploit balancing across RX queues, the IXIA was configured to generate 64 bytes TCP packets (minimum packet size) originated from a single IP address towards a rotating set of 4096 IP destination addresses. With 64 bytes packets, a full Gbit link can carry up to 1.48 Million of packets per second (Mpps).

Table 1: Evaluation platforms

	low-end	**high-end**
motherboard	Supermicro PSDBE	Supermicro X8DTL-iF
CPU	Core2Duo 1.86 Ghz 2 cores 0 HyperThreads	2x Xeon 5520 2.26 Ghz 8 cores 8 HyperThreads
Ram	4 GB	4 GB
NIC	Intel ET (1 Gbps)	Intel ET (1 Gbps)

In order to perform performance measurements we used *pfcount*, a simple traffic monitoring application that counts the number of captured packets. Depending on the configuration, pfcount spawns multiple packet capture threads per network interface and even concurrently capture from multiple network devices, including *virtual capture devices*.

In all tests we enabled multi-queues in drivers, and modified the driver's code so that queue information is propagated up to PF_RING; such driver does not spawn any poller thread at the kernel level, and does not avoid socket buffer allocation. We call this driver MQ (multi-queue) and TNAPI the one described in Section 3.

Comparing Different Approaches: As a first test, we evaluated the packet capture performance when using multi-threaded packet capture applications with and without multi-queue enabled. To do so, we measured the maximum loss free rate when pfcount uses three different two-threaded setups:

- *Setup A*: multiple queues are disabled and therefore capture threads read packets from the same interface (single queue, SQ). Threads are synchronized using a r/w semaphore. This setup corresponds to the default Linux configuration shown in Figure 1.
- *Setup B*: two queues are enabled (MQ) and there are two capture threads consuming packets from them. No synchronization is needed.
- *Setup C*: there is one capture thread at the user level and a polling thread at the kernel level (TNAPI).

Table 2: Packet capture performance (kpps) at 1 Gbps with different two-threads configurations.

		Setup A	**Setup B**	**Setup C**
Platform	SQ	SQ	MQ	TNAPI
	Threads Userspace/Kernel space			
	1/0	2/0	2/0	11
low-end	721 Kpps	640 Kpps	610 Kpps	1264 Kpps
high-end	1326 Kpps	1100 Kpps	708 Kpps	1488 Kpps

Table 2 shows the performance results on the multi-threaded setups, and also shows as a reference point the single-threaded application. The test confirmed the issues we described in Section 2. When pfcount spawns two threads at the user level, the packet capture performance is actually worse than the single-threaded one. This is expected in both cases (setup A and B). In case of setup A, the cause of the drop compared to the single-threaded setup is cache invalidations due to locking (semaphore), whereas for B the cause is the round robin IRQ balancing. On the other hand, our approach consisting of using a kernel thread and a thread at the user level (setup C) is indeed effective and allows the low-end platform to almost double its single-thread performance. Moreover, the high-end machine can capture 1 Gbps (1488 kpps) with no loss.

CPU Affinity and Scalability: We now turn our attention to evaluating our solution at higher packet rates with the high-end platform. We are interested in understanding if by properly setting the CPU affinity is possible to effectively partition the computing resources and therefore increase the maximum loss-free rate.

2 NICS: To test the packet capture technology with more traffic, we plug another Intel ET NIC into the high-end systems and we inject with the IXIA traffic generator 1.488 Mpps for each interface (wire-rate at 1 Gbit with 64 bytes packets). We want to see if it is possible to handle 2 full Gbit links with two cores and two queues per NIC only. To do so, we set the CPU affinity to make sure that for every NIC the two polling threads at the kernel level are executed on different Hyper-Threads belonging to the same core (e.g. 0 and 8 from Figure 4 belong to Core 0 of the first physical processor[4]). We use the pfcount application to

[3]See pthread_setaffinity_np().

[4]Under Linux, */proc/cpuinfo* lists the available processing units and report for each of them the core identifier and the physical CPU. Processing units sharing the same physical CPU and core identifier are Hyper-Threads.

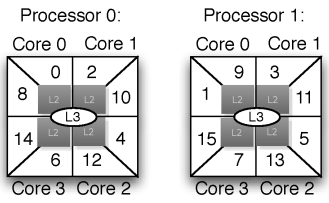

Figure 4: Core Mapping on Linux with the Dual Xeon. Hyper-Threads on the same core (e.g. 0 and 8) share the L2 cache.

Table 3: Packet capture performance (kpps) when capturing concurrently from two 1 Gbit links.

Test	Capture threads affinity	Polling threads affinity	NIC1 Kpps	NIC2 Kpps
1	not set	not set	1158	1032
2	NIC1@0 on 0 NIC1@1 on 8 NIC2@0 on 2 NIC2@1 on 10	NIC1@0 on 0 NIC1@1 on 8 NIC2@0 on 2 NIC2@1 on 10	1122	1290
3	NIC1 on 0,8 NIC2 on 2,10	NIC1@0 on 0 NIC1@1 on 8 NIC2@0 on 2 NIC2@1 on 10	1488	1488

spawn capture threads and we perform measurements with three configurations. First of all, we measure the packet capture rate when one capture thread and one polling thread per queue is spawn (8 threads in total) without setting the CPU affinity (Test 1). Then (Test 2), we repeat the test by binding each capture thread to the same Hyper-Thread where the polling thread for that queue is executed (e.g. for the queue NIC1@0 both polling and capture thread run on Hyper-Thread 0). Finally, in Test 3, we reduce the number of capture thread to one for each interface. Each capture thread runs on the same core where polling threads for the NIC is capturing from are executed.

Table 3 reports the maximum loss-free rate when capturing from two NIC simultaneously using the configurations previously described. As shown in Test 1, without properly tuning the system by means of CPU affinity, our test platform is not capable to capture at wire rate from two adapters simultaneously. Test 2 and Test 3 show that the performance can be substantially improved by setting the affinity and wire rate achieved. In fact, by using a single capture thread for each interface (Test 3) all incoming packets are captured with no loss (1488 kpps per NIC).

In principle, we would expect to achieve the wire rate with the configuration in Test 2 rather than the one used in Test 3. However, splitting the load on two RX queues means that capture threads are idle most of the time, at least on high-end processors such as the Xeons we used and a dummy application that only counts packets. As a consequence, capture threads must call *poll()* very often as they have no packet to process and therefore go to sleep until a new packet arrives; this may lead to packet losses. As system calls are slow, it is better to keep capture threads busy so that poll() calls are reduced. The best way of doing so is to capture from two RX queues, in order to increase the number of incoming packets. It is worth noting that, since monitoring applications are more complex than pfcount, the configuration used for Test 2 may provide better performance in practice.

4 NICS: We decided to plug two extra NICs to the system to check if was possible to reach the wire-rate with 4 NICs at the same time (4 Gbps of aggregated bandwidth with minimum sized packets). The third and forth NIC were configured using the same tuning parameters as in Test 3 and the measurements repeated. *The system can capture 4 Gbps of traffic per physical processor without losing any packet.*

Due to lack of NICs at the traffic generator we could not evaluate the performance at more than 4 Gbps with synthetic streams of minimum size packets representing *the worst-case scenario for a packet capture technology.* However, preliminary tests conducted on a 10 Gbit production network (where the average packet size was close to 300 bytes and the used bandwidth around 6 Gbps) confirmed that this setup is effective in practice.

The conclusion of the validation is that when CPU affinity is properly tuned, our packet technology allows:

- Packet capture rate to scale linearly with the number of NICs.
- Multi-core computers to be partitioned processor-by-processor. This means that load on each processor does not affect the load on other processors.

5. RELATED WORK

The industry followed three paths for accelerating network monitoring applications by means of specialized hardware while keeping the software flexibility. Smart traffic balancers, such as cPacket[1], are special purpose devices used to filter and balance the traffic according to rules, so that multiple monitoring stations receive and analyze a portion of the traffic. Programmable network cards [5] are massively parallel architectures on a network card. They are suitable for accelerating both packet capture and traffic analysis, since monitoring software written in C can be compiled for that special purpose architecture and run on the card and not on the main host. Unfortunately, porting applications on those expensive devices is not trivial. Capture accelerators [2] completely offload monitoring workstations from the packet capturing task leaving more CPU cycles to perform analysis. The card is responsible to copy the traffic directly to the address space of the monitoring application and thus the operating system is completely bypassed.

Degiovanni and others [9] show that first generation packet capture accelerators are not able to exploit the parallelism of multi-processor architectures and propose the adoption of a software scheduler to increase the scalability. The scalability issue has been solved by modern capture accelerators that provide facilities to balance the traffic among several threads of execution. The balancing policy is implemented by their firmware and it is not meant to be changed at run-time as it takes seconds if not minutes to reconfigure.

The work described in [18] highlights the effects of cache coherence protocols in multi-processor architectures in the context of traffic monitoring. Papadogiannakis and others [21] show how to preserve cache locality for improving traffic analysis performance by means of traffic reordering.

Multi-core architectures and multi-queue adapters have been exploited to increase the forwarding performance of

software routers [13, 14]. Dashtbozorgi and others [8] propose a traffic analysis architecture for exploiting multi-core processors. Their work is orthogonal to ours, as they do not tackle the problem of enhancing packet capture through parallelism exploitation.

Several research efforts show that packet capture can be substantially improved by customizing general purpose operating systems. nCap [11] is a driver that maps the card memory in user-space, so that packets can be fetched from user-space without any kernel intervention. The work described in [25] proposes the adoption of large buffers containing a long queue of packets to amortize the cost of system calls under Windows. PF_RING [10] reduces the number of packet copies, and thus, increases the packet capture performance, by introducing a memory-mapped channel to carry packets from the kernel to the user space.

6. OPEN ISSUES AND FUTURE WORK

This work represents a first step toward our goal of exploiting the parallelism of modern multi-core architectures for packet analysis. There are several important steps we intend to address in future work. The first step is to introduce a software layer capable of automatically tuning the CPU affinity settings. The task, which is crucial for achieving high performance, is not yet trivial for non expert users.

In addition, one of the basic assumption of our technology is that the hardware-based balancing mechanism (RSS in our case) is capable of evenly distributing the incoming traffic across cores. This is often, but not always, true in practice. In the future, we plan to exploit mainstream network adapters supporting hardware-based and dynamically configurable balancing policies [12] for implementing an adaptive hardware-assisted software packet scheduler that is able to dynamically distribute the workload among cores.

7. CONCLUSIONS

This paper highlighted several challenges when using multi-core systems for network monitoring applications: resource competition of threads on network buffer queues, unnecessary packet copies, interrupt and scheduling imbalances. We proposed a novel approach to overcome the existing limitations and showed solutions for exploiting multi-cores and multi-queue adapters for network monitoring. The validation process has demonstrated that by using TNAPI it is possible to capture packets very efficiently both at 1 and 10 Gbit. Therefore, our results present the first software-only solution to show promise towards offering scalability with respect to the number of processors for packet capturing applications.

8. ACKNOWLEDGEMENT

The authors would like to thank J. Gasparakis and P. Waskiewicz Jr from Intel^{TM}for the insightful discussions about 10 Gbit on multi-core systems, as well M. Vlachos, and X. Dimitropoulos for their suggestions while writing this paper.

9. CODE AVAILABILITY

This work is distributed under the GNU GPL license and is available at no cost from the ntop home page http://www.ntop.org/.

10. REFERENCES

[1] cpacket networks - complete packet inspection on a chip. http://www.cpacket.com.

[2] Endace ltd. http://www.endace.com.

[3] Ixia leader in converged ip testing. Homepage http://www.ixiacom.com.

[4] Libpcap. Homepage http://www.tcpdump.org.

[5] A. Agarwal. The tile processor: A 64-core multicore for embedded processing. Proc. of HPEC Workshop, 2007.

[6] K. Asanovic et al. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[7] A. Cox. Network buffers and memory management. The Linux Journal, Issue 30,(1996).

[8] M. Dashtbozorgi and M. Abdollahi Azgomi. A scalable multi-core aware software architecture for high-performance network monitoring. In *SIN '09: Proc. of the 2nd Int. conference on Security of information and networks*, pages 117–122, New York, NY, USA, 2009. ACM.

[9] L. Degioanni and G. Varenni. Introducing scalability in network measurement: toward 10 gbps with commodity hardware. In *IMC '04: Proc. of the 4th ACM SIGCOMM conference on Internet measurement*, pages 233–238, New York, NY, USA, 2004. ACM.

[10] L. Deri. Improving passive packet capture: beyond device polling. Proc. of SANE, 2004.

[11] L. Deri. ncap: Wire-speed packet capture and transmission. In *E2EMON '05: Proc. of the End-to-End Monitoring Techniques and Services*, pages 47–55, Washington, DC, USA, 2005. IEEE Computer Society.

[12] L. Deri, J. Gasparakis, P. Waskiewicz Jr, and F. Fusco. Wire-Speed Hardware-Assisted Traffic Filtering with Mainstream Network Adapters. In *NEMA'10: Proc. of the First Int. Workshop on Network Embedded Management and Applications*, page to appear, 2010.

[13] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, L. Mathy, and P. Papadimitriou. A platform for high performance and flexible virtual routers on commodity hardware. *SIGCOMM Comput. Commun. Rev.*, 40(1):127–128, 2010.

[14] N. Egi, A. Greenhalgh, M. Handley, G. Iannaccone, M. Manesh, L. Mathy, and S. Ratnasamy. Improved forwarding architecture and resource management for multi-core software routers. In *NPC '09: Proc. of the 2009 Sixth IFIP Int. Conference on Network and Parallel Computing*, pages 117–124, Washington, DC, USA, 2009. IEEE Computer Society.

[15] F. Fusco, F. Huici, L. Deri, S. Niccolini, and T. Ewald. Enabling high-speed and extensible real-time communications monitoring. In *IM'09: Proc. of the 11th IFIP/IEEE Int. Symposium on Integrated Network Management*, pages 343–350, Piscataway, NJ, USA, 2009. IEEE Press.

[16] Intel. Accelerating high-speed networking with intel i/o acceleration technology. White Paper, 2006.

[17] Intel. Intelligent queuing technologies for

virtualization. White Paper, 2008.

[18] A. Kumar and R. Huggahalli. Impact of cache coherence protocols on the processing of network traffic. In *MICRO '07: Proc. of the 40th Annual IEEE/ACM Int. Symposium on Microarchitecture*, pages 161–171, Washington, DC, USA, 2007. IEEE Computer Society.

[19] R. Love. Cpu affinity. Linux Journal, Issue 111,(July 2003).

[20] B. Milekic. Network buffer allocation in the freebsd operating system. Proc. of BSDCan,(2004).

[21] A. Papadogiannakis, D. Antoniades, M. Polychronakis, and E. P. Markatos. Improving the performance of passive network monitoring applications using locality buffering. In *MASCOTS '07: Proc. of the 2007 15th Int. Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 151–157, Washington, DC, USA, 2007. IEEE Computer Society.

[22] L. Rizzo. Device polling support for freebsd. BSDConEurope Conference, (2001).

[23] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the bandwidth wall: challenges in and avenues for cmp scaling. *SIGARCH Comput. Archit. News*, 37(3):371–382, 2009.

[24] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond softnet. In *ALS '01: Proc. of the 5th annual Linux Showcase & Conference*, pages 18–18, Berkeley, CA, USA, 2001. USENIX Association.

[25] M. Smith and D. Loguinov. Enabling high-performance internet-wide measurements on windows. In *PAM'10: Proc. of Passive and Active Measurement Conference*, pages 121–130, Zurich, Switzerland, 2010.