Exploiting Commodity Multi-core Systems for Network Traffic Analysis

Luca Deri

ntop.org
Pisa, Italy
deri@ntop.org

Francesco Fusco

IBM Zurich Research Laboratory
Rüschlikon, Switzerland
ffu@zurich.ibm.com

Abstract

The current trend in computer processors is towards multi-core systems. Although operating systems were adapted a long time ago to support multi-processing, kernel network layers have not yet taken advantage of this new technology. The result is that packet capture, the cornerstone of every network monitoring application, is not efficient on modern systems and its performance gets worse with an increasing number of cores.

This paper describes common pitfalls of network monitoring applications when used with multi-core systems, and presents solutions to these problems. In addition, it covers the design and implementation of a new multi-core aware packet capture kernel module that enables monitoring applications to scale with the number of cores, contrary to what happens in most operating systems.

Keywords: Passive packet capture, multi-core processors, network traffic monitoring, Linux kernel, operating systems.

1. Introduction

The complexity of Internet-based services and advances in interconnection technologies increased the demand for advanced monitoring applications designed for high-speed networks. The increased complexity of monitoring tasks such as anomaly detection, intrusion detection and traffic classification, made software extremely attractive because it is more flexible and less expensive than dedicated hardware. On the other hand, analyzing high-speed network by means of software applications running on commodity off-the-shelf (COTS) hardware presents major performance challenges. Packet capture is still one of the most resource intensive tasks for the majority of passive monitoring applications. The industry followed three paths for accelerating software applications by means of specialized hardware while keeping the software flexibility:

- Accelerating the packet capture process in hardware via packet capture accelerators [8].
- Splitting the traffic among different monitoring stations [24] in order to reduce the application workload.
- Developing programmable hardware boards with standard languages [25, 26].

Along with hardware-based solutions, researchers [15, 21, 27, 28] have demonstrated that packet capture performance of software solutions based on commodity hardware can be substantially improved by enhancing general purpose operating systems for traffic analysis. Today's COTS hardware offers features (and performance) that just a few years ago were only provided by expensive special purpose hardware: desktop-class machines are becoming advanced multi-core (or even multi-processor) parallel architectures [1, 2, 22] capable of concurrently execute multiple threads at the same time. Modern network adapters feature several independent transmission (TX) and reception (RX) queues, each mapped on a separate core [9]. Initially designed for facilitating the implementation of virtual machine supervisors, network queues can also be used to accelerate network traffic tasks, such as routing [31] by processing incoming packets into concurrent threads of execution. Unfortunately, operating systems are not yet able to fully take advantage of this breakthrough network technology for network traffic analysis, thus dramatically limiting its scope of application.

In this paper, we introduce a new packet capture technology named TNAPI, which exploits the increased parallelism of commodity hardware for speeding up network analysis applications. The rest of the paper is structured as follow. Section 2 provides an analysis of multi-core technology when applied to packet capture. Section 3 describes the common design pitfalls when developing network analysis application on top of multi-core. Section 4 describes TNAPI, and its performance is evaluated in Section 5. Finally, section 6 lists some open issues and future development activities.

2. Packet Capture and Commodity Multi-core Systems

Exploiting parallel architectures to perform passive network analysis is a challenge for several reasons. First, packet capture applications are memory bound, but memory bandwidth does not seem to increase as fast as the number of core available [14]. Second, balancing the traffic among different processing units is challenging, as it is not possible to predict the nature of the incoming traffic. Exploiting the parallelism with general-purpose operating systems is even more difficult as they have not been designed for accelerating packet capture. During the last three decades, memory access has always been one of the worst enemies of scalability and thus several solutions to this problem have been proposed. With the advent of symmetric multiprocessor systems (SMP), multiple processors are connected to the same memory bus, hereby causing processors to compete for the same memory bandwidth. Integrating the memory controller inside the main processor is another approach for increasing the memory bandwidth. The main advantage of this architecture is fairly obvious: multiple memory modules can be attached to each processor, thus increasing bandwidth. However this architecture also has some disadvantages. Since the memory has to be shared between all the processors, the memory access is no longer uniform (hence the name NUMA - Non Uniform Memory Access). Local memory (i.e. the memory close to a processor) can be accessed quickly, whereas the situation is completely different when a processor accesses a memory close to another one. In shared memory multiprocessors, such as SMP and NUMA, a cache coherence protocol [3] must be used in order to guarantee synchronization among processors. A multi-core processor is a processing system composed of two or more or more individual processors, called cores, integrated onto a single chip package. As a result, the inter-core bandwidth of multi-core processors can be many times greater than the one of SMP systems. In addition inter-core latencies are substantially reduced. In some cases, the number of processor cores is increased by

leveraging technologies such as hyper-threading that allows a physical core to be partitioned even further. Cores often share components such as the level-2 cache and the front side bus, so they cannot usually be regarded as truly independent processors.
For traffic monitoring applications, the most effective approach to optimize the bandwidth utilization is to reduce the number of packet copies. With general purpose operating systems, the journey of the packet inside the kernel can be long and it usually involves at least two copies: from the network card to a socket buffer, and from the socket buffer to the monitoring application. The packet capture performance can be substantially increased by reducing this journey using memory map [15]. Capture accelerators provide a straight path from the wire to the address space of the monitoring application. Thus, the operating system overhead is completely avoided. However, bandwidth utilization can also be optimized by increasing the cache locality of packet capture application.
The work in [6] shows that improving the cache locality of packet capture software through packet reordering allows the overall performance to be significantly improved. However, this approach is unfeasible with high speed networks as packet reordering is very costly. In order to better exploit the cache subsystem, Intel introduced some dedicated circuitry to assist the task of handling incoming packets, known as Direct Cache Access (DCA) [4, 10], which allows incoming packets to be written directly into the core's cache. In multi-core and multi-processor architectures, memory bandwidth can be wasted [18] in many ways, including improper scheduling, wrong balancing of interrupt (IRQ) requests, and subtle mechanisms such as false sharing [20, 29]. For these reasons, squeezing performance out of those architectures requires additional effort. Even though there is a lot of ongoing research in this area [9], most of the existing schedulers are unaware of architectural differences between cores, therefore the scheduling does not guarantee the best memory bandwidth utilization. This may happen when two threads using the same data set are scheduled on different processors, or on the same multi-core processors having separate cache domains. What happens in this case is that the two processors (or cores) fetch the same data from the memory, i.e, the same data is fetched twice. When cache levels are shared among cores this situation can be avoided by placing both threads on the same multi-core processor. Unfortunately, schedulers are not so sophisticated and thus, the intervention of the programmer through the manipulation of CPU affinity [23], is required in order to ensure that scheduling is more effective. The work described in [13] shows that the introduction of a software scheduler to better distribute the workload among threads can substantially increase the scalability. The same work also demonstrated that the increased parallelism can justify the cost of synchronization between threads.

Capture accelerators supporting multiple ring buffers implement in firmware the logic for balancing the traffic according to traffic rules, so that different processes or threads receive and analyze a portion of the traffic. It is worth noting that the balancing scheme, although programmable, is not meant to be modified at runtime and, in fact, a rule set reconfiguration may take seconds, if not minutes. Thanks to this feature, capture accelerators smoothed the transition towards parallel network packet processing. Similar technologies, such as Receive Side Scaling (RSS) [7], have also been introduced in modern off-the-shelf network interface cards to increase the networking throughput on multi-core machines. RSS enabled network adapters include the logic for balancing incoming traffic across multiple RX queues. This balancing policy is implemented in hardware and thus RSS is not as flexible as rule based systems. However, this simple policy is effective in practice also for network monitoring applications, as most of them are flow-oriented. To the best of our knowledge, none of the available packet capture software is able of fully benefiting from RSS technology. In summary, we have identified the following issues while engineering monitoring applications for off-the-shelf multi-core and multi-processor architectures:

- The operating system scheduler is completely unaware of the workload and in some cases it does not have the knowledge to relocate threads on the right core/processors.
- Balancing the interruptions among different cores may not be an appropriate strategy.
- Balancing the workload among processors is not straightforward, as the workload depends on the incoming traffic, which cannot be predicted.
- Preserving the cache locality is a prerequisite in order to achieve a good scalability on modern parallel architectures and to overcome the bandwidth limitations, but monitoring software has poor cache locality.

- Modern parallel architectures are getting complex and their performance is hard to predict. Optimizing the code for parallel architectures may be much more difficult than optimizing code for uniprocessor architectures. Cache coherence has to be taken into account during the implementation.
- As general purpose operating systems are not optimized for network monitoring, packet capture performance, the cornerstone of every monitoring applications, do not fully take benefit from the increased parallelism and from the advanced features offered by modern network adapters.

The contribution of our work is a new packet capture technology based on a general purpose operating system, designed for exploiting the parallelism of commodity parallel architecture for network traffic analysis. The technology allow to achieve high packet capture rates on those architectures without requiring extensive knowledge, and provide the basic mechanisms for dividing the workload among different processing units. For this reason, we believe it can smooth the transition of monitoring applications toward parallel traffic analysis.

3. Common Application Design Pitfalls

As previously explained, modern multi-core-aware network adapters are logically divided in several RX/TX queues where packets are flow-balanced across queues using hardware-based facilities such as Intel RSS (Receive-side Scaling) part of Intel I/O Acceleration Technology (I/O Acceleration Technology) [5, 10]. As this technology has been designed for virtualization, it is not commonly used in network traffic monitoring. The following figure highlights some design limitations of many packet capture applications, that could have been fixed by exploiting RSS that logically partitions a network adapter into multiple virtual RX/TX queues. The number of such queues depends on the NIC chipset, and it is limited by the number of cores. This means that on a given NIC, RSS on a dual-core machine can create up to two RX queues per port, whereas RSS can virtualize four RX queues per port when using the same NIC on a quad-core machine.

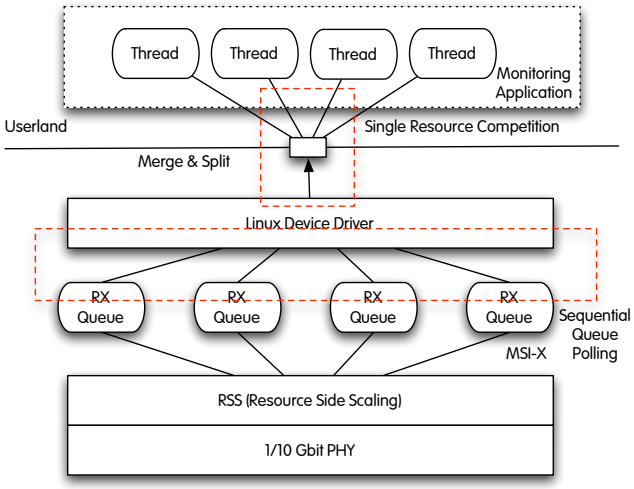

Figure 1. Design Limitations in Network Monitoring Architectures

In network monitoring, it is very important to make sure that all incoming packets are captured and forwarded to the monitoring applications. Modern network adapters are trying to improve network performance by splitting a single RX queue into several queues, each mapped to a processor core. The idea is to balance the load, both in terms of packets and interrupts, across all cores hence to improve the overall performance. Device drivers are unable to preserve this design up to the application: they merge all queues in one as it used to happen with legacy adapters featuring only one queue. This limitation is a major performance bottleneck, because even if a userland application uses 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, as happens at the kernel level. In multi-core systems, this problem is even worse because it is not often possible to map the monitoring application on the same core from which packets are coming. In addition the use of semaphores that, as a side effect, invalidates the processor's cache, which represents the basic ingredient for

preserving multi-core performance. In a nutshell, current network layer design needs to "merge and split" packets a couple of times and access them using semaphores, instead of providing a straight, lock-less path to the application with no performance limitation due to cache invalidation.

In most operating systems, packets are fetched using packet polling [11,12] techniques that have been designed in the pre-multi-core age, when network adapters had only one 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 a result, it is not possible to poll RX queues in parallel but only sequentially, nor is possible that packets coming from queue X are marked as such and have this information delivered to the user space. The latter information could be profitably used for balancing traffic inside the monitoring application. In summary, modern network adapters offer several features that are not exploited at software level as many applications sequentially read packets from a single source instead of reading them in parallel without locks from all the RX queues simultaneously. Moreover, these limitations also have an effect on cache usage because they prevent applications from being bound to the same core from which packets are coming.
Another area where performance can be improved is related to memory management. In network monitoring, packets are often received on dedicated adapters not used for routing or management; this means that they do not need to be forwarded nor routed but just used for monitoring. Additionally, in most operating systems, captured packets are moved to userland via mapped memory [15] rather than using system calls such as read () that are much less efficient. This means that it is possible to use zero-copy techniques to carry packets from the kernel to user-space. Unfortunately, incoming packets are copied into a memory area [16, 17] that holds the packet until it gets copied to userland via memory map. This causes unnecessary kernel memory allocation and deallocation, as zero-copy could start directly at the driver layer and not just at the networking layer. The problem of efficient memory management is serious for multi-core systems because in addition to overheads due to allocation/deallocation, operating systems do not usually feature efficient multi-core-aware memory allocators [19] resulting in serializing memory allocation across cores/threads hence further reducing application performance.

4. Towards Efficient Multi-core-aware Monitoring Architectures

So far this paper has identified some features of multi-core systems that need to be taken into account while designing applications, and it has also highlighted how operating systems do not fully take advantage of modern network interface cards. The enhancements previously discussed have been introduced to the Linux kernel by modifying both the networking layer and device drivers. This is necessary to poll RX queues independently and to let this information propagate up to the userland where monitoring applications are executed. Linux has been selected as the target kernel because it is the de-facto reference platform for the evaluation of novel packet capture technologies. However, all the concepts are general and can also be adapted to other operating systems. Instead of modifying a vanilla kernel, the authors enhanced a home-grown Linux kernel module named PF_RING [15] and modified the underlying network device drivers. In order to unleash the driver performance, all available RX queues are polled simultaneously by kernel threads (one kernel thread per queue), hence the term TNAPI (Threaded NAPI). As shown in figure 2, TNAPI replaces Linux NAPI [30] packet polling mechanism with a thread per RX queue, that fetches packets from the NIC and pushes them to the networking stack. Depending on the PF_RING configuration, packets can follow the standard journey into the kernel or be pushed directly to PF_RING. Beside the speed advantage offered by PF_RING with respect to vanilla Linux, it allows the incoming RX queue id to be preserved. As this information is used by PF_RING to implement balancing across packet consumer application, throughout this paper TNAPI is always used on top of PF_RING. Another advantage of TNAPI when used on top of PF_RING derives from memory management. As both the NIC RX ring and PF_RING RX ring are allocated statically, TNAPI does not copy packets into socket buffers (skb) as vanilla Linux does; instead it copies the packet payload from the RX NIC ring top the PF_RING RX ring. This means that for each incoming packet, costly skb memory allocation/deallocation is avoided as well memory mapping across the PCI bus. Finally, if TNAPI sends directly packets to PF_RING, it can refrain from passing packets to PF_RING whenever the PF_RING RX, this to preserve CPU cycles.

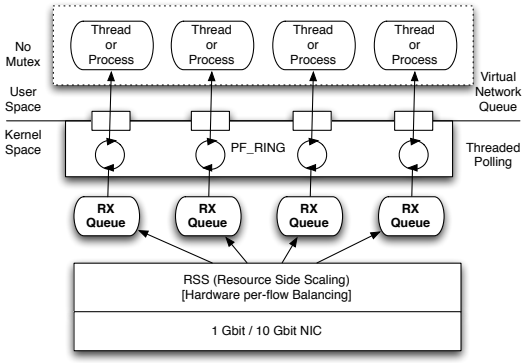

Figure 2. TNAPI (Threaded NAPI)

Contrary to PF_RING that is a network layer on top of NIC drivers, TNAPI is implemented directly into the NIC driver as it is necessary to change the mechanism that notifies the kernel when an incoming packet has been received. As of today, TNAPI is available for modern Intel 1 and 10 Gbit network adapters (82575/6 and 82598/9 chipsets) that support RSS hence multiple RX/TX queues. The TNAPI driver is responsible of spawning one thread per RX queue per port, and binding it to the same core where interrupts for such queue are received. PF_RING has been adapted to read the queue identifier from TNAPI before upper layers invalidate its value. In such a way the authors implemented inside PF_RING the concept of "virtual network adapter", a feature provided only by high-end capture accelerator. 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. The latter solution allows applications to be easily split into several independent threads of execution, each receiving and analyzing a portion of the traffic. As previously explained, in order to avoid performance degradation due to suboptimal cache utilization, it is highly recommended to bind the thread or process to the same core to which the virtual interface/queue is bound. This operation is easily doable by manipulating the CPU affinity. The following section will explain CPU affinity tuning in more in detail.

5. TNAPI Evaluation

In this section the performance and scalability of TNAPI is compared to PF_RING. The authors did not port TNAPI to other operating systems such as the BSD family, as the idea behind validation is to position TNAPI with respect to PF_RING and vanilla linux packet capture, and not to compare it with other operating systems. The authors evaluated the work using two different parallel architectures belonging to different market segments (low-end and high-end). The first monitoring station is based on a single Core 2 processor (model E6300) running at 1.86 GHz and equipped with 2 MB of L2 cache shared among the cores. The second one is a high-end machine equipped with two Xeon i7 processors (model E5520) running at 2.27 GHz and a Supermicro X8DTL based server board. Each processor is equipped with four hyper-threaded cores sharing 6 MB of L3 cache, for a total of 16 cores. Both the monitoring stations are equipped with 4 GB of RAM and an Intel ET 1 Gbit installed on a 4x PCI-E slot. The latter platform supports the latest features introduced by Intel such as DCA and I/OAT, whereas the Core2Duo board does not. An IXIA 400 has been used to generate the network traffic for experiments. In order to exploit balancing across RX queues, the IXIA was configured to generate 64 bytes TCP packets originated from a single IP address towards a rotating set of 4096 IP destination addresses.

The first test measured the packet capture performance of simple packet capture-and-discard (i.e. no packet processing is performed) application named *pfcount*. PF_RING is used both on top of vanilla NIC drivers and PF_RING-aware drivers; in the latter case, the driver sends packets directly to PF_RING without using the Linux stack (this is called transparent mode), and then discards the packet. Finally pfcount has been tested on top of PF_RING with TNAPI.

Figure 3 depicts the maximum loss-free packet capture rate of pfcount in various configurations when increasing the number of capture threads. The input packet rate is 1.488 Mpps with 64 bytes long packets (wire speed with minimum size packets). This test has been repeated in a few configurations:

- PF_RING in 'vanilla' mode (i.e. it hooked to the Linux kernel as AF_PACKET)
- PF_RING in 'transparent' mode (i.e. packets are pushed by the NIC driver to PF_RING without using the standard kernel mechanisms).
- Single Queue (SQ): the NIC driver is the one that comes with the Linux kernel and it does not enable multiple RX queues into the NIC (default behavior on Linux).
- Multiple Queue (MQ): the NIC driver is PF_RING-aware (i.e. it is part of the PF_RING source tree) so that it sends directly to PF_RING incoming packets by also passing the RX queue identifier on which the packet has been received. In this configuration the driver is started in multi-queue mode.
- PF_RING+TNAPI: PF_RING is started in transparent mode, on top of the NIC driver started in MQ mode.

Note that in SQ mode, a single multi-threaded pfcount per NIC is activated. In MQ mode, two multi-threaded pfcount instances have been activated, each polling from a different queue; in this case the sum of packets captured by both application has been reported.

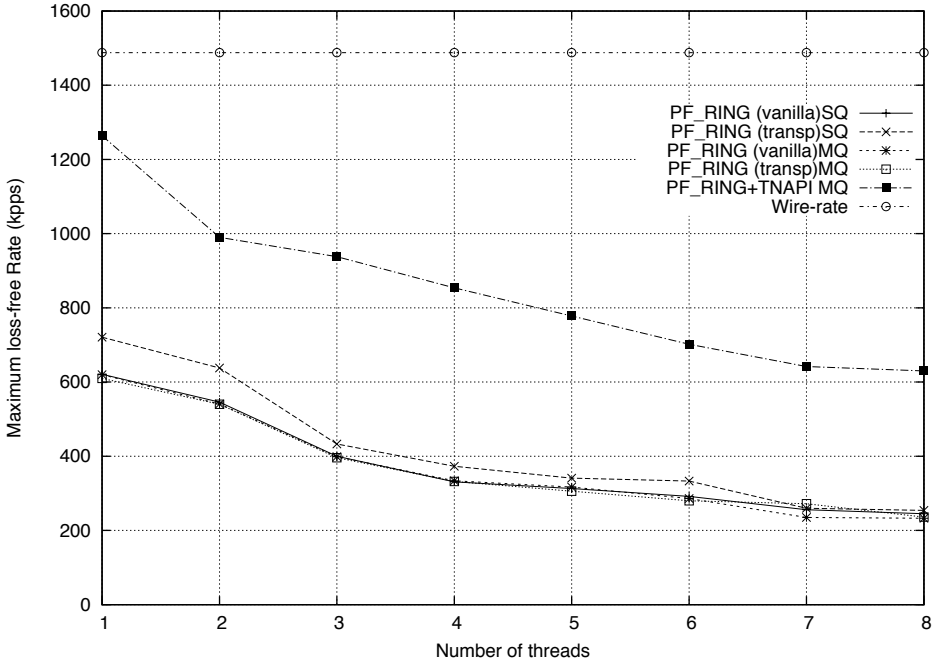

Figure 3. TNAPI and PF_RING Performance Tests: Captured Packets vs. Number of Threads

As shown in Figure 3, the original PF_RING does not benefit from multi-queue adapters. In fact, the performance is roughly the same when a single or multiple queues are enabled. Spawning more packet capture threads does not provide any performance gain, and, on the contrary, is the cause of a performance drop. This is obvious for thread pools larger than the number of physical cores (greater than 2) and expected also for the two thread scenario, as they compete for a shared resource and spend most of their time in synchronizations. A similar behavior can be observed when TNAPI is enabled. In fact by using two threads (the TNAPI kernel thread and the packet capture thread at the user level) the application can reach 1260 Kpps which is close, but not equal to the wire-rate. On the other side, by spawning two threads at user-level (TNAPI disabled) the maximum loss-free rate is 540 Kpps which is even lower than the one obtained with a single one thread (610 Kpps).

So far, this test has demonstrated that contrary to the original PF_RING, TNAPI allows the parallelism of the Core2Duo processor to be exploited.

Test Platform	PF_RING SQ	PF_RING MQ	PF_RING+TNAPI
Core2Duo (pfcount not bound to a CPU)	721 Kpps	610 Kpps	1264 Kpps
Dual i7 (pfcount not bound to a CPU)	1326 Kpps	708 Kpps	1488 Kpps
Dual i7 (pfcount bound to a CPU)	858 Kpps	1178 Kpps	1488 Kpps

Table 1. TNAPI and PF_RING Performance Tests Using pfcount (mono-threaded)

Table 1 reports the results of pfcount mono-threaded on the two selected test platforms. The test has demonstrated that:

- When TNAPI is enabled, the Core2Duo is performing as good as the high-end platform equipped with the original version of PF_RING.
- TNAPI allows the high-end platform to capture packets at wire-rate performance.
- Multi-queue technology is not beneficial for packet capture unless properly used, as its use can result in major performance drops with respect to single queue.
- Binding (also known as SMP affinity) pfcount to a specific CPU (by means of the taskset tool) can significantly change the performance figures.

In addition to SMP affinity, it is important also to take into account how IRQ balancing affects results. In fact, the NIC sends an interrupt whenever it has to notify the operating system that a new packet has been received. With MQ, the operating system assigns a different IRQ to each RX queue. In order to better understand how SMP affinity and IRQ balancing affects the performance figures hence TNAPI scalability, the authors decided to play with various setups on the dual i7 platform. Even if NICs support up to 8 RX queues, the authors decided to configure only two RX queues, as they are enough for handling 1 Gbit of traffic. Before reporting the test results, it is worth to understand how i7‘s NUMA affects the tests.

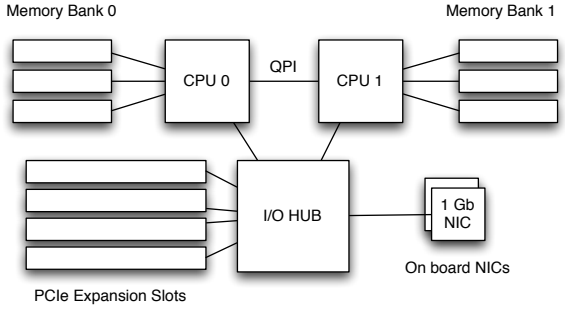

Figure 4. Typical NUMA Architecture

In NUMA architectures, memory is local to a CPU and when a core needs to access memory not allocated on the same CPU, it cannot access it directly but it must go through QPI (Quick Path Interconnect). The same applies to IRQs that for maximum performance must be delivered to the same core on which the application that consumes such interrupts resides. This means that in order to optimize the performance, for a given RX queue it is necessary to bind on the same core:

- The TNAPI thread that fetches packets from the NIC adapter.
- The IRQ used for the queue.

In Linux `/proc/cpuinfo` lists the available CPUs. Each entry contains:

- processor: unique core identifier
- physical id: the identifier of the CPU on which the core resides.
- core id: the unique identifier of the core inside the CPU is running on.

Processor	Physical CPU	CoreId	Processor	Physical CPU	CoreId
0	0	0	**8**	0	0
1	1	0	**9**	1	0
2	0	1	**10**	0	1
3	1	1	**11**	1	1
4	0	2	**12**	0	2
5	1	2	**13**	1	2
6	0	3	**14**	0	3
7	1	3	**15**	1	3

Table 2. Cores Mapping on Linux with Dual i7

The table 2 shows that processors with an adjacent processor identifier are not physically close. Furthermore due to Hyper-Threading, processor 0 and 8 are basically two twin cores, similar to 1 and 9, and so on. Table 3 reports the maximum packet capture rate when capturing from two NIC simultaneously while the traffic generator is injecting traffic at wire speed. In all tests interrupts are bound onto the same core used for TNAPI: for NIC 1 processors 1 and 9 are used, for NIC 2 processors 2 and 10 are used.

Test	pfcount Affinity	TNAPI Threads Affinity	NIC 1	NIC 2
1	0-15	0-15	1158	1032
2	0-15	NIC1 queues on 1,9 NIC2 queues on 2,10	945	1088
3	pfcount on NIC1 queues on 1,9 pfcount on NIC2 queues on 2,10	NIC1 queues on 1,9 NIC2 queues on 2,10	1090	1200
4	pfcount on NIC1@0 on 1 pfcount on NIC1@1 on 9 pfcount on NIC2@0 on 2 pfcount on NIC2@1 on 10	NIC1@0 on 1 NIC1@1 on 9 NIC2@0 on 2 NIC2@1 on 10	1122	1290
5	pfcount on NIC1 queues on 1,9 pfcount on NIC2 queues on 2,10	NIC1@0 on 1 NIC1@1 on 9 NIC2@0 on 2 NIC2@1 on 10	1488	1488

Table 3. How SMP affinity and IRQ Balancing Affect Packet Capture Throughput (Kpps)

The test outcome shows that, even with TNAPI, a very fast machine as a dual i7 is not able to capture at wire speed from two adapters simultaneously, unless the affinity is properly tuned. In fact, setting the wrong CPU affinity may cause a substantial drop of the aggregated packet capture rate (i.e., the difference between test 2 and test 5 is over 900 Mpps). In principle, the pfcount application should be bound to the same core used for the RX queue it handles. However splitting the load on two RX queues means that pfcount is idle most of its time, at least on fast processors as the i7. As a consequence pfcount must call poll() very often as it has no packet to process hence it needs to go to sleep until new packet arrive; this may lead to packet losses. As system calls are slow, it is better to keep pfcount busy so that poll() calls are reduce. The best way of doing so is to bind pfcount to two RX queues, this in order to increase the number of incoming packets. In fact, in test 5 pfcount has been able to capture all incoming packets with no loss (i.e. 1.48 Mpps). Note that if pfcount is replaced with a more computing intensive application (hence that does not call poll() too often) settings of test 4 may provide better performance.

The authors decided to plug two extra NICs to the system to check weather was possible to reach the wire-rate with 4 NICs at the same time (4 Gbit/s of aggregated bandwidth). The 3rd and 4th NIC were configured using the same tuning parameter as in test 5 and the measurements repeated. All four pfcount applications captured the traffic with no loss. Preliminary tests at 10 Gbit confirmed that this setup is also effective on this scenario, with the difference that the card used in the tests (based on chipset 82598) does not support more than eight RX queues, so scalability beyond 4 Gbit needs to be verified on newer adapters (based on chipset 82599).

The conclusion of TNAPI validation is that SMP, IRQ, TNAPI and pfcount correct binding 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.

In a nutshell, if incoming traffic can be balanced across all available processors, in principle (this because authors have not tested it beyond 4 Gbit due to lack of NICs on the traffic generator) TNAPI can scale linearly with the number of NICs and processors.

6. Open Issues and Future Work Items

One of the basic assumptions of multi-core systems is the ability to balance load across cores. Modern network adapters have been designed to share the load across RX queues using technologies such as RSS. The basic assumption is that incoming traffic can be balanced, that is often true but not all the time. In this case, a few cores will be responsible for handling all incoming packets whereas other cores will be basically idle. This problem is also present in TNAPI that takes advantage of RSS for traffic balancing. The authors are currently investigating software solutions whose aim is to further balance the traffic on top of RSS by creating virtual RX queues in addition to the physical ones, while preserving the actual performance. This problem is the same as balancing the traffic at the kernel level on top of legacy network adapters that do not feature multiple RX queues.

7. Conclusions

The use of multi-cores enables the development of both high-speed and memory/computing-intensive applications. The market trend is clearly towards many-core architectures as they are the only ones able to provide developers with a positive answer to network monitoring needs, as well as provide the scalability for future high-speed networks. This paper has highlighted some of the challenges users face when using multi-core systems for the purpose of network monitoring. Although multi-core is the future of computing, it is a technology that needs to be used with care to avoid performance issues as well as, in some cases, performance degradation. The authors have identified which aspects need to be taken into account when developing applications for multi-core systems, as well as the limitations of current monitoring architectures. TNAPI is a novel approach to overcome existing operating systems limitations and unleash the power of multi-core when used 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, and in contrast to the current operating systems generation, it can scale almost linearly with the number of processors.

Code Availability

This work is distributed under the GNU GPL2 license and is available at http://www.ntop.org/TNAPI.html.

8. References

[1] P. Van Roy, The Challenges and Opportunities of Multiple Processors: Why Multi-Core Processors are Easy and Internet is Hard, Proceedings of ICMC, (2008).

[2] J. Frühe, Planning Considerations for Multi-core Processor Technology, White Paper, (2005).

[3] T. Tartalja and V. Milutinovich, The Cache Coherence Problem in Shared-Memory Multiprocessors: Software Solutions, ISBN: 978-0-8186-7096-1, (1996).

[4] A. Kumar and R. Huggahalli, Impact of Cache Coherence Protocols on the Processing of Network Traffic, 40th Annual IEEE/ACM International Symposium on Microarchitecture ,(2007).

[5] Intel, Intelligent Queuing Technologies for Virtualization, White Paper, (2008).

[6] A. Papagiodannakis and others, Improving the Performance of Passive Network Monitoring Applications using Locality Buffering, Proceedings of MASCOTS, (2007).

[7] Microsoft, Scalable Networking: Eliminating the Receive Processing Bottleneck Introducing RSS, WinHEC (Windows Hardware Engineering Conference) (2004).

[8] A. Heyde, Investigating the performance of Endace DAG monitoring hardware and Intel NICs in the context of Lawful Interception, CAIA Technical Report 080222A, (2008).

[9] Intel, Improving Network Performance in Multi-Core Systems, White Paper, (2007).

[10] Intel, Accelerating High-Speed Networking with Intel I/O Acceleration Technology, White Paper, (2006).

[11] J. Salim and R. Olsson, Beyond Softnet, Proceedings of the 5th annual Linux Showcase & Conference, (2001).

[12] L. Rizzo, Device Polling Support for FreeBSD, BSDConEurope Conference, (2001).

[13] L. Degioanni and G. Varenni, Introducing scalability in network measurement: toward 10 Gbps with commodity hardware, Internet Measurement Conference, (2004).

[14] K. Asanovic and others, The landscape of parallel computing research: A view from Berkley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkley, December (2006).

[15] L. Deri, Improving Passive Packet Capture: Beyond Device Polling, Proceedings of SANE, (2004).

[16] A. Cox, Network Buffers and Memory Management, The Linux Journal, Issue 30, (1996).

[17] B. Milekic, Network Buffer Allocation in the FreeBSD Operating System, Proceedings of BSDCan, (2004).

[18] C. Leiserson and I. Mirman, How to Survive the Multi-core Software Revolution, Cilk Arts, (2009).

[19] S. Schneider, Scalable Locality-Conscious Multithreaded Memory Allocation, Proceedings of the ACM SIGPLAN, (2006).

[20] H. Sutter, Eliminate False Sharing, Dr. Dobb's Journal, Issue 5, (2009).

[21] F. Fusco and others, Enabling High-Speed and Extensible Real-Time Communications Monitoring, 11th IFIP/IEEE International Symposium on Integrated Network Management, (2009).

[22] T. Sterling, Multi-Core for HPC: breakthrough or breakdown?, Panel of SC06 Conference, (2006).

[23] R. Love, Linux System Programming: Talking Directly to the kernel and C Library, O'Reilly Media Inc., (2007)

[24] R. Kay, Pragmatic Network Latency Engineering Fundamental Facts and Analysis, White Paper, (2009).

[25] D. Comer, Network systems design using network processors, Computing Reviews. Vol. 45, no. 9, (2004).

[26] A. Agarwal, The Tile processor: A 64-core multicore for embedded processing, Proceedings of HPEC Workshop, (2007).

[27] M. Dashtbozorgi and M. Abdollahi Azgomi, A high-performance software solution for packet capture and transmission, Proceedings of ICCSIT, (2009).

[28] L. Deri, nCap: Wire-speed Packet Capture and Transmission, Proceedings of E2EMON, (2005).

[29] W.J. Bolosky and M. L. Scott, False sharing and its effect on shared memory performance, USENIX Experiences with Distributed and Multiprocessor Systems, (1994).

[30] J. Hadi Salim and others, Beyond Softnet, Proceedings of Usenix Annual Technical Conference, (2001).

[31] M. Dobrescu and others, RouteBricks: Exploiting Parallelism To Scale Software Routers, 22nd ACM Symposium on Operating Systems Principles (SOSP), (2009).