

Exploiting Commodity Multicore Systems for Network Traffic Analysis

Luca Deri

ntop
Pisa, Italy
Email: deri@ntop.org

Francesco Fusco

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

Abstract—The current trend in computer processors is towards multicore systems. Although operating systems have been adapted 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 system and its performance gets worse with the number of cores.

This paper describes common pitfalls of network monitoring applications when used with multicore systems, and presents solutions to these problems. In addition, it covers the design and implementation of a new multicore-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, multicore processors, traffic monitoring, Linux kernel.

I. INTRODUCTION

The computer industry constantly demands better products able to increase performance while reducing costs. The processor clock speed race started in the 90s, and ended when the processor manufacturers realized that this race also produced an increase in processor complexity and power consumption as well heat dissipation problems. Fortunately, faster processors are not the only way to increase performance. This is because better performance can also be achieved by reducing processor clock rate while increasing the number of computing units. Contrary to early symmetric multiprocessing (SMP) systems in which only discrete processors were used, the basic idea behind multicore is to partition the processor into independent processor units, known as cores, that are integrated on a single chip package. Processor 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. In some cases, the number of processor cores is increased by leveraging technologies such as hyperthreading that allows a physical processor to be partitioned into a few logical processors, each capable of simultaneously executing a thread of an application.

Unfortunately multicore technology has so far been available only for the main computer processor. Network traffic analysis did not really benefit from this technology, except in terms of increased computational power. Only in the past couple of years has the industry realized that in order to speed-up network traffic analysis it is necessary to apply the

“divide and conquer” principle not just on processors but also on network interface adapters. Modern multicore-aware network adapters feature several independent RX/TX 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 improve network traffic analysis by processing incoming packets into concurrent threads of execution; this results in a significantly faster analysis speed with respect to traditional single-threaded packet analysis. Unfortunately operating systems are not yet able to fully take advantage of this breakthrough network technology, thus dramatically limiting its scope of application.

The aim of this paper is to highlight the benefits of multicore technology on commodity hardware when applied to network traffic analysis. In particular it explains how existing network monitoring applications can be adapted in order to fully exploit multicore systems. In addition, it shows the limitations of modern operating systems design when dealing with modern network adapters, and covers the design and implementation of a novel packet capture technique that fully takes advantage of them, resulting in superior network traffic analysis performance.

II. UNDERSTANDING MULTICORE SYSTEMS

Exploiting the parallelism of current architectures in order to perform passive network traffic monitoring is a challenge for several reasons. In fact most of the monitoring applications are memory bound, and memory bandwidth does not seem to increase as fast as the number of cores available [14]. During the last three decades, memory accesses always represented the worst enemy of scalability and thus several solutions to this problems have been proposed. With the advent of symmetric multiprocessor systems, 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 pretty clear: 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 must be used in order to guarantee synchronization among processors. Multicore processors can be seen as conventional symmetric multiprocessors, though they offer new capabilities which makes them fundamentally different from traditional symmetric multiprocessors. This is because:

- the inter-core bandwidth of multicore can be many times greater than the one of SMP systems;
- inter-core latencies are substantially reduced;
- multicore allows cache levels to be shared on the same die.

Sharing cache levels on the same die may be important for multicore processors, as their scalability is affected by the same memory bandwidth problem. Given that memory bandwidth issues will probably affect current and future parallel architectures [20], optimizing the cache utilization is fundamental. The work referenced in [6] shows that improving the cache locality of packet capture software through packet reordering allows the overall performance to be dramatically improved. Unfortunately this approach is unfeasible with high speed networks as packet reorder is very costly. In order to better exploit the cache subsystem, Intel introduced some dedicated circuitry to assist the task of handling incoming packets. The idea of Direct Cache Access (DCA) [10] is to extend the protocols between the network interface card, memory controller and processors so that the network interface has the ability to write the received packet directly into the core's cache. The usual solution for improving the cache utilization is to group data together to reduce the working set and make caching much more effective. Following this approach with cache coherent architectures may result in wasting memory bandwidth due to cache trashing [18]. In a nutshell cache trashing may happen when the memory is accessed in a way that invalidates cache lines resulting in excessive cache misses. These misses are the cause of an incredible performance drop. Cache trashing may be caused by the cache coherency protocol. In fact, due to the cache coherency protocol, modifying a shared cache line involves the invalidation of other processors' caches and generates further access to the memory. Thus, memory bandwidth can be saved by grouping read-only variables and read-write variables in different cache lines. In this way, situations like false sharing [20] can be avoided.

Memory bandwidth may also be wasted by improper scheduling. Even though there is a lot of ongoing research in this area [9], most of the existing schedulers are basically unaware of architectural differences between cores, so that the scheduling does not guarantee the best utilization of the memory bandwidth. This may happen when two threads using the same data set are scheduled on different processors or on the same multicore processors having separate cache domains. What happens in this case is that the two processors (or cores) fetch the same data from the memory, so that 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 multicore 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 done efficiently.

Balancing the workload among different processing units is important for exploiting the parallelism. Keeping the workload balanced is challenging by itself, because it is not possible to predict anything on incoming traffic patterns. Bursty traffic, for example, can limit the scalability of hardware accelerated packet capture solutions. Capture accelerators such as Endace DAG cards [8], are special purpose network cards designed to efficiently transfer packets from wire to the address space of the monitoring application by means of a circular buffer. The work described in [13] shows that the introduction of a software scheduler for better distributing the workload among threads, can substantially increase the scalability. This also proved 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 they behave like programmable packet dispatchers. In this way, the problem identified in [13] is solved in a much more effective way. For that reason capture accelerators smoothed the transition to wards parallel network packet processing. DCA can be used in conjunction with RSS technology [7], a hardware packet balancer across network adapters RX queues, in order to distribute the workload among different cores hence increase the cache utilization. However, it is worth to remark that both solutions have some limitations. With RSS, the balancing algorithm is implemented in hardware, thus it cannot be modified. With capture accelerators, the balancing scheme is implemented in firmware so that users can customize the dispatching behavior. However, modifying the dispatching behavior requires a card reconfiguration and may take seconds if not minutes, and thus the balancing scheme is not meant to be modified at runtime.

In summary we identified some issues while porting monitoring applications on multicore architectures:

- The 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 the proper strategy.
- Balancing the workload among processors is not trivial, 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 architecture are getting complex and their performance is hard to predict. Optimizing the code for a parallel architecture may be much more difficult than optimizing code for a uniprocessor architecture. Cache coherence has to be taken into account during the implementation.

III. COMMON APPLICATION DESIGN PITFALLS

As previously explained, modern multicore-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's RSS part of Intel I/OAT (I/O Acceleration Technology) [21]. Unfortunately this property is

used mostly for virtualization and not for network traffic monitoring. The following figure highlights some design limitations of the lack of exploitation of the above technologies when applied to monitoring architectures.

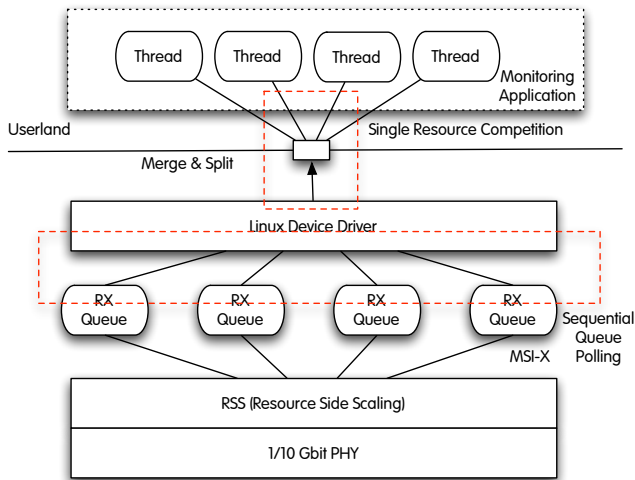


Figure 1. Design Limitations in Network Monitoring Architectures

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. This preventing that a single core is overwhelmed, by dedicating a private RX queue to each core. Unfortunately device drivers are unable to preserve this design up to the application: they merge all queues in one as happened with legacy adapters featuring only one queue. This limitation is a major performance bottleneck, as 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 other techniques have to be used in order to serialize this work instead of carrying it on in parallel, as it happens at the kernel level. In multicore 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. Not to mention the use of semaphores that have as side effect processor’s cache invalidation that represents the basic ingredient for preserving multicore performance. In a nutshell, antiquate network layer design requires 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-multicore age when network adapters had only one RX queue. This means that 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 the card details. This means that it is not possible to poll RX queues in parallel but only sequentially, nor that packets coming from queue X are marked as such and have this information float up to the userland. The latter information could be profitably used for balancing traffic

inside the monitoring application. The result is that hardware cards offer several features that are not exploited at software level. The consequence is that network applications are not as fast as they could be, and also that these architectural limitations force applications to be coded in a sub-optimal way by forcing them to sequentially read packets from a single source instead of reading them in parallel without locks from all the RX queues simultaneously. Finally 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 move a packet 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 into the driver and not just at the networking layer. The problem of efficient memory management is very important on multicore systems as in addition to overhead due to allocation/deallocation, operating systems do not usually feature efficient multicore-aware memory allocators [19] resulting in serializing memory allocation across cores/threads hence further reducing application performance.

IV. TOWARDS EFFICIENT MULTICORE-AWARE MONITORING ARCHITECTURES

So far this paper has identified some features of multicore 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. As previously discussed, it has been necessary to modify both the kernel networking layer and device drivers. This is necessary to poll RX queues independently and to let this information propagate until the userland where monitoring applications are executed. Instead of modifying a vanilla kernel, the authors used a home-grown Linux kernel module named PF_RING [15] and adapt the underlying network device drivers. In order to validate the proposed solution at various network speed and conditions, the authors decided not to limit this work at 1 Gbit but also to validate at 10 Gbit where monitoring applications are experiencing major performance challenges when using today’s operating systems. This validation work has been implemented on Linux but all the concepts are general and can be adapted also to other operating systems.

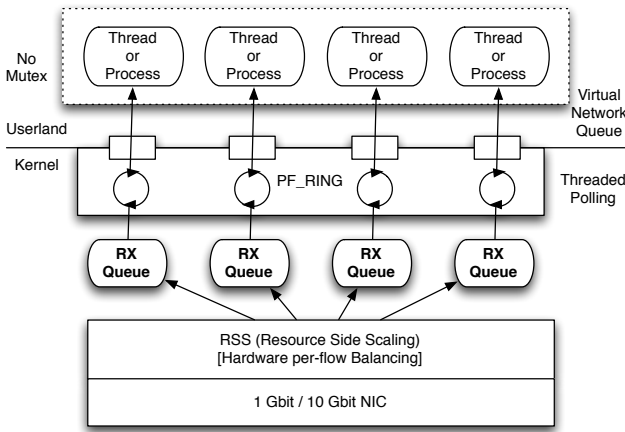


Figure 2. TNAPI (Threaded NAPI)

In order to unleash the driver performance, Linux NAPI has been modified so that all available RX queues are polled simultaneously by kernel threads (one kernel thread per queue), hence the term TNAPI (Threaded NAPI). In order to achieve this, when the driver starts-up it spawns one thread per RX queue and binds it to the same core where interrupts are received. The standard NAPI polling mechanism is then disabled as the threads push packets to the networking layer as they arrive. TNAPI can be used with vanilla Linux kernels or in combination with PF_RING. In the latter case there are several advantages as PF_RING has been optimized for working with TNAPI. In fact TNAPI, when used with PF_RING, has the ability to know whether PF_RING memory is full, this due to incoming packets not yet consumed by the monitoring application, hence to stop reading packets. When not used with PF_RING, in order not to let threads monopolize the processor pushing packets up the stack, TNAPI periodically yields the processor to leave other tasks.

Another change to the driver has been the propagation of the queue identifier up to the networking layer. Unfortunately Linux upper network layers are not able to view the card as split into several rings but just as a single entity. Thus, the queue identifier is lost during the packet journey to userland. For this reason PF_RING, that is sitting at the bottom of the network layer, has been adapted to read the queue identifier from TNAPI before that upper layers invalidate its value. This means that inside PF_RING it has been implemented the concept of virtual network adapter. 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 receiving only packets received by a specific queue. The latter solution allows applications to be split into several independent threads of execution, each receiving a portion of the packets. In case an application has been bound to a specific queue it is important to also bind the application to the same core to which the queue is bound, this to avoid performance degradation due to suboptimal cache usage as previously explained.

Performance issues due to network memory buckets allocation/deallocation in particular on multicore systems, have been overcome when TNAPI is used with PF_RING. In

this case TNAPI allocates network buckets only when the adapter is activated. Whenever an incoming packet is received, instead of unmapping this bucket and allocating a new one for future incoming packets, it does copy the bucket content into a free PF_RING slot. This can be done because PF_RING allocates its memory as a single kernel memory area to which incoming packets are copied, contrary to the vanilla kernel where packets need to be allocated in memory areas that are then queued on a linked list.

In order to prevent the problem of false sharing, PF_RING data-structures shared between the kernel and userland have been rearranged so that they are now both aligned with the cache lines and read/write and read-only variables are grouped together.

In order to evaluate TNAPI and PF_RING performance, the authors used an IXIA 400 traffic generator, Intel ET 1 Gbit and Intel AF 10 Gbit cards, a mainboard supporting up to two Xeon processors, and a pair of Intel Xeon 2.4 GHz quad-core CPUs (5400 series). In order to exploit balancing across RX queues, test traffic had to be easy to balance. For this reason 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 authors did not port TNAPI to other operating systems such as the BSD family, as the idea behind validation is to show that TNAPI provides a major performance advantage with respect to vanilla Linux, and not to compare Linux with other operating systems.

In order to evaluate the scalability, the performance using one or two processor has been compared. The packet capture performance (in packets per second) has been evaluated using a simple packet capture application named *pfcount* implemented on top of PF_RING. Initially *pfcount* has been tested at 1 Gbit with a single processor installed. Then tests have been repeated with TNAPI with single and multi-RX queues. In this case one TNAPI kernel thread is spawn per RX queue. It is worth to remark that *pfcount* can be configured to either spawn one or more packet capture threads per device/queue. However during the tests only one polling thread was used as this resulted in better performance than the multithreaded version, this due to the use of semaphores for serializing access to the PF_RING socket from which packets are received.

TABLE I. TNAPI AND PF_RING PERFORMANCE TESTS

Max Packet Capture Speed	
Test Type	I/OAT
PF_RING (No TNAPI) 1 processors, 1 queue	620K pps (1 Gbit)
PF_RING (No TNAPI) 2 processors, 1 queue	760K pps (1 Gbit)
PF_RING + TNAPI (single RX queue) 1 processors	920K pps (1 Gbit)
PF_RING + TNAPI (Multi RX queue) 1 processor, 4 queues	Wire Rate (1 Gbit) ~3 Mpps (10 Gbit)
PF_RING + TNAPI (Multi RX queue) 2 processor, 8 queues	Wire Rate (1 Gbit) ~5 Mpps (10 Gbit)

The results show that due to the reduction of memory allocations and the introduction of the kernel thread, TNAPI allows the performance to be almost doubled when compared to PF_RING without TNAPI; this even when a single RX queue is enabled. When multiple queues are enabled, a single processor is sufficient for capturing packets at 1 Gbit wire-rate without any packet loss. Then, the same tests have been repeated at 10 Gbit, with either one or two processors installed. In this case, adding the second processor doubled the original four RX queues, increasing the performance 3 Mpps to 5 Mpps, resulting in a scalability which is close to 80%.

In conclusion, the validation has demonstrated that:

- PF_RING performance does not scale well when doubling the number of processors, unless TNAPI is used.
- Doubling the number of processors, the overall capture performance increased of 80% when using both PF_RING and TNAPI.

V. OPEN ISSUES AND FUTURE WORK ITEMS

One of the basic assumptions of multicore 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 times. 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 additional virtual RX queues in addition to the physical ones, while preserving the actual performance. This problem is the same as providing in kernel traffic balancing on top of legacy network adapters that do not feature multiple RX queues.

VI. FINAL REMARKS

The use of multicores enables the development of both high-speed and memory/computing-intensive applications. The market trend is clearly towards multicore solutions as they are the only ones able to provide developers 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 multicore systems for the purpose of network monitoring. Although multicore 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 multicore systems, as well the limitations of current monitoring architectures. TNAPI is a novel approach to overcome existing operating systems limitations and unleash the power of multicore when used for network monitoring. The validation process has demonstrated that 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 GPL 2 license and is available at the ntop home page <http://www.ntop.org/>.

ACKNOWLEDGMENT

The authors would like to thank Alexander Komarov <alexander.komarov@intel.com>, Endace Measurement Systems, and ntop users for their comments and suggestions.

REFERENCES

- [1] Peter Van Roy, The Challenges and Opportunities of Multiple Processors: Why Multi-Core Processors are Easy and Internet is Hard, 2008.
- [2] J. Frühe, Planning Considerations for Multicore Processor Technology, White Paper, May 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, March 2007.
- [5] Intel, Intelligent Queueing 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] Stephen Donnelly, DAG Packet Capture Performance, White Paper, August 2006.
- [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] Loris Degioanni, Gianluca Varenni, Introducing scalability in network measurement: toward 10 Gbps with commodity hardware, Internet Measurement Conference, 2004.
- [14] K. Asanovic, R. Bodik, B. Catanzaro, et al. 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] Alan Cox, Kernel Korner: 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 Multicore 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. Dobbs's Journal, Issue 5, (2009).
- [21] Intel, Accelerating High-Speed Networking with Intel I/O Acceleration Technology, White Paper, 2006.
- [22] T. Sterling, Multi-core issues - Multi-Core for HPC: breakthrough or breakdown?, Panel of SC06 Conference, 2006.
- [23] Robert Love, Linux System Programming: Talking Directly to the kernel and C Library, O'Reilly Media Inc., 2007