Improving the Performance of Passive Network Monitoring Applications with Memory Locality Enhancements

Antonis Papadogiannakis[a,*], Giorgos Vasiliadis[a], Demetres Antoniades[a], Michalis Polychronakis[b], Evangelos P. Markatos[a]

[a] *Institute of Computer Science, Foundation for Research and Technology – Hellas P.O.Box 1385 Heraklion, GR-711-10 Greece*
[b] *Computer Science Department, Columbia University, New York, USA*

Abstract

Passive network monitoring is the basis for a multitude of systems that support the robust, efficient, and secure operation of modern computer networks. Emerging network monitoring applications are more demanding in terms of memory and CPU resources due to the increasingly complex analysis operations that are performed on the inspected traffic. At the same time, as the traffic throughput in modern network links increases, the CPU time that can be devoted for processing each network packet decreases. This leads to a growing demand for more efficient passive network monitoring systems in which runtime performance becomes a critical issue.

In this paper we present *locality buffering*, a novel approach for improving the runtime performance of a large class of CPU and memory intensive passive monitoring applications, such as intrusion detection systems, traffic characterization applications, and NetFlow export probes. Using locality buffering, captured packets are being reordered by clustering packets with the same port number before they are delivered to the monitoring application. This results in improved code and data locality, and consequently, in an overall increase in the packet processing throughput and decrease in the

*Corresponding author. Tel: +30 2810 391663. Fax: +30 2810 391601.
Email addresses: papadog@ics.forth.gr (Antonis Papadogiannakis), gvasil@ics.forth.gr (Giorgos Vasiliadis), danton@ics.forth.gr (Demetres Antoniades), mikepo@cs.columbia.edu (Michalis Polychronakis), markatos@ics.forth.gr (Evangelos P. Markatos)

packet loss rate. We have implemented locality buffering within the widely used `libpcap` packet capturing library, which allows existing monitoring applications to transparently benefit from the reordered packet stream without modifications. Our experimental evaluation shows that locality buffering improves significantly the performance of popular applications, such as the Snort IDS, which exhibits a 21% increase in the packet processing throughput and is able to handle 67% higher traffic rates without dropping any packets.

Keywords: passive network monitoring, intrusion detection systems, locality buffering, packet capturing

---

1. Introduction

Along with the phenomenal growth of the Internet, the volume and complexity of Internet traffic is constantly increasing, and faster networks are constantly being deployed. Emerging highly distributed applications, such as media streaming, cloud computing, and popular peer-to-peer file sharing systems, demand for increased bandwidth. Moreover, the number of attacks against Internet-connected systems continues to grow at alarming rates.

As networks grow larger and more complicated, effective passive network monitoring is becoming an essential operation for understanding, managing, and improving the performance and security of computer networks. Passive network monitoring is getting increasingly important for a large set of Internet users and service providers, such as ISPs, NRNs, computer and telecommunication scientists, security administrators, and managers of high-performance computing infrastructures.

While passive monitoring has been traditionally used for relatively simple network traffic measurement and analysis applications, or just for gathering packet traces that are analyzed off-line, in recent years it has become vital for a wide class of more CPU and memory intensive applications, such as network intrusion detection systems (NIDS) [1], accurate traffic categorization [2], and NetFlow export probes [3]. Many of these applications need to inspect both the headers and the whole payloads of the captured packets, a process widely known as *deep packet inspection* [4]. For instance, measuring the distribution of traffic among different applications has become a difficult task. Several recent applications use dynamically allocated ports, and therefore, cannot be identified based on a well known port number. Instead, protocol parsing and several other heuristics, such as searching for an application-

specific string in the packets payload [2], are commonly used. Also, intrusion detection systems such as `Snort` [1] and `Bro` [5] need to be able to inspect the actual payload of network packets in order to detect malware and intrusion attempts. Threats are identified using attack "signatures" that are evaluated by advanced pattern matching algorithms.

The complex analysis operations of such demanding applications incur an increased number of CPU cycles spent on every captured packet. Consequently, this reduces the overall processing throughput that the application can sustain without dropping incoming packets. At the same time, as the speed of modern network links increases, there is a growing demand for more efficient packet processing using commodity hardware that is able to keep up with higher traffic loads.

A common characteristic that is often found in passive monitoring applications is that they usually perform different operations on different types of packets. For example, a NIDS applies a certain subset of attack signatures on packets with destination port 80, *i.e.*, it applies the web-attack signatures on packets destined to web servers, while a different set of signatures is applied on packet destined to database servers, and so on. Furthermore, NetFlow probes, traffic categorization, as well as TCP stream reassembly, which has become a mandatory function of modern NIDS [6], all need to maintain a large data structure that holds the active network flows found in the monitored traffic at any given time. Thus, for packets belonging to the same network flow, the process accesses the same part of the data structure that corresponds to the particular flow.

In all above cases, we can identify a *locality* of executed instructions and data references for packets of the same type. In this paper, we present a novel technique for improving packet processing performance by taking advantage of this locality property which is commonly exhibited by many different passive monitoring applications. In practice, the captured packet stream is a mix of interleaved packets that correspond to hundreds or thousands of different packet types, depending on the monitored link. Our approach, called *locality buffering*, is based on reordering the packet stream that is delivered to the monitoring application in a way that enhances the locality of the application's code execution and data access, improving the overall packet processing performance.

We have implemented locality buffering in `libpcap` [7], the most widely used packet capturing library, which allows for improving the performance of a wide range of passive monitoring applications written on top of `libpcap`

in a transparent way, without the need to modify them. Our implementation combines locality buffering with memory mapping, which optimizes the performance of packet capturing by mapping the buffer in which packets are stored by the kernel into user level memory.

Our experimental evaluation using real-world applications and network traffic shows that locality buffering can significantly improve packet processing throughput and reduce the packet loss rate. For instance, the popular Snort IDS exhibits a 21% increase in the packet processing throughput and is able to process 67% higher traffic rates with no packet loss.

The rest of this paper is organized as follows: Section 2 outlines several methods that can be used for improving packet capturing performance. In Section 3 we describe the overall approach of locality buffering, while in Section 4 we present in detail our implementation of locality buffering within the `libpcap` packet capturing library combined with memory mapping. Section 5 presents the experimental evaluation of our prototype implementation using three popular passive monitoring tools. Finally, Section 6 summarizes related work, Section 7 discusses limitations of our approach and future work and Section 8 concludes the paper.

2. Background

Passive monitoring applications analyze network traffic by capturing and examining individual packets passing through the monitored link, which are then analyzed using various techniques, from simple flow-level accounting, to fine-grained operations like deep packet inspection. Popular passive network monitoring applications, such as intrusion detection systems, per-application traffic categorization, and NetFlow export probes, are built on top of libraries for generic packet capturing. The most widely used library for packet capturing is `libpcap` [7].

2.1. Packet Capturing in Linux

We briefly describe the path that packets take from the wire until they are delivered to the user application for processing. All packets captured by the network card are stored in memory by the kernel. In Linux [8], this is achieved by issuing an interrupt for each packet or an interrupt for a batch of packets [9, 10]. Then, the kernel hands the packets over to every socket that matches the specified BPF filter [11]. In case that a socket buffer becomes full, the next incoming packets will be dropped from this socket. Thus,

the size of the socket buffer affects the tolerance of a passive monitoring application in short-term traffic or processing bursts. Finally, each packet is copied to memory accessible by the user-level application.

The main performance issues in the packet reception process in Linux that affect the packet capturing and processing throughput are the following:

- *High interrupt service overhead* (per packet cost): Even a fast processor is overwhelmed by constantly servicing interrupts at high packet arrival rates, having no time to process the packets (*receive livelock*) [9]. NAPI [10] combines polling with interrupts to solve this problem.
- *Kernel-to-user-space context switching* (per packet cost): It takes place when a packet crosses the kernel-to-user-space border, *i.e.*, calling a system call for each packet reception. Thus, the user level packet processing will start several milliseconds later. Using a memory mapped buffer between kernel and user space for storing packets solves this problem efficiently, since packets are accessible directly from user space without calling any system calls.
- *Data copy and memory allocation costs* (per byte cost): Copying the packet data from NIC to kernel memory and from kernel-level to user-level memory consumes a significant amount of CPU time. Zero-copy approaches [12, 13, 14] have been proposed to reduce such costs.

Braun *et al.* [15] and Schneider *et al.* [16] compare the performance of several packet capturing solutions on different operating systems using the same hardware platforms and provide guidelines for system configuration to achieve optimal performance.

2.2. Using Memory Mapping between Kernel and User-level Applications

In order to avoid kernel-to-user-space context switches and packet copies from kernel to user space for each captured packet, a memory-mapped ring buffer shared between kernel and user space is used to store the captured packets. The general principle of memory-mapping is to allow access from both kernel and user space to the same memory segment. The user level applications are then able to read the packets directly from the ring buffer, avoiding context switching to the kernel.

The ring buffer plays the same role as the socket buffer that we described earlier. The kernel is capable of inserting packets captured by the network interface into the ring buffer, while the user is able to read them directly from

there. In order to prevent race conditions between the two different processes, an extra header is placed in front of each packet to ensure atomicity while reading and writing packets into the buffer. Whenever the processing of a packet is over, it is marked as read using this header, and the position in which the packet is stored is considered by the kernel as empty. The kernel uses an *end* pointer that points to the first available position to store the next arrived packet, while the user-level application uses a *start* pointer that points to the first non-read packet. These two pointers guarantee the proper operation of the circular buffer: The kernel simply iterates through the circular buffer, storing newly arrived packets on empty positions and blocks whenever the end pointer reaches the last empty position, while the user application processes every packet in sequence as long as there are available packets in the buffer. The main advantage of the memory mapped circular buffer is that it avoids the context switches from kernel to user level for copying each packet. The latest versions of Linux kernel and `libpcap` support this memory mapping functionality, through the `PACKET_MMAP` option that is available in the `PACKET` socket interface.

Packets are still being copied from the DMA memory allocated by the device driver to the memory mapped ring buffer through a software interrupt handler. This copy leads to a performance degradation, so *zero copy* techniques [13, 14, 17, 18, 19] have been proposed to avoid it. These approaches can avoid this data copy by sharing a memory buffer between all the different network stack layers within kernel and between kernel and user space.

2.3. Using Specialized Hardware

Another possible solution to accelerate packet capturing is to use specialized hardware optimized for high-speed packet capture. For instance, DAG monitoring cards [20] are capable of full packet capture at high speeds. Contrary to commodity network adapters, a DAG card is capable of retrieving and mapping network packets to user space through a *zero-copy* interface, which avoids costly interrupt processing. It can also stamp each packet with a high precision timestamp. A large static circular buffer, which is memory-mapped to user-space, is used to hold arriving packets and avoid costly packet copies. User applications can directly access this buffer without the invocation of the operating system kernel.

When using DAG cards, the performance problems occurred in Linux packet capturing can be eliminated, but at a price that is prohibitively high for many organizations. On the other hand, commodity hardware is always

preferable and much easier to find and deploy for network monitoring. In addition, specialized hardware alone may not be enough for advanced monitoring tasks at high network speeds, *e.g.*, intrusion detection.

3. Locality Buffering

The starting point of our work is the observation that several widely used passive network monitoring applications, such as intrusion detection systems, perform almost identical operations for a certain class of packets. At the same time, different packet classes result to the execution of different code paths, and to data accesses to different memory locations. Such packet classes include the packets of a particular network flow, *i.e.*, packets with the same protocol, source and destination IP addresses, and source and destination port numbers, or even wider classes such as all packets of the same application-level protocol, *e.g.*, all HTTP, FTP, or BitTorrent packets.

Consider for example a NIDS like Snort [1]. Each arriving packet is first decoded according to its Layer 2–4 protocols, then it passes through several *preprocessors*, which perform various types of processing according to the packet type, and finally it is delivered to the main inspection engine, which checks the packet protocol headers and payload against a set of attack signatures. According to the packet type, different preprocessors may be triggered. For instance, IP packets go through the IP defragmentation preprocessor, which merges fragmented IP packets, TCP packets go through the TCP stream reassembly preprocessor, which reconstructs the bi-directional application level network stream, while HTTP packets go through the HTTP preprocessor, which decodes and normalizes HTTP protocol fields. Similarly, the inspection engine will check each packet only against a subset of the available attack signatures, according to its type. Thus, packets destined to a Web server will be checked against the subset of signatures tailored to Web attacks, FTP packets will be checked against FTP attack signatures, and so on.

When processing a newly arrived packet, the code of the corresponding preprocessors, the subset of applied signatures, and all other accessed data structures will be fetched into the CPU cache. Since packets of many different types will likely be highly interleaved in the monitored traffic mix, different data structures and code will be constantly alternating in the cache, resulting to cache misses and reduced performance. The same effect occurs in other monitoring applications, such as NetFlow collectors or traffic clas-

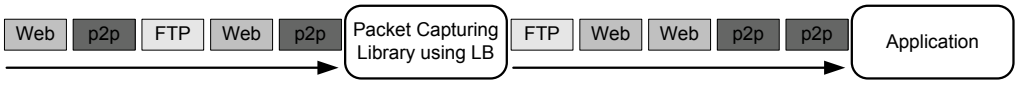

Figure 1: The effect of locality buffering on the incoming packet stream.

sification applications, in which arriving packets are classified according to the network flow in which they belong to, which results to updates in a corresponding entry of a hash table. If many concurrent flows are active in the monitored link, their packets will arrive interleaved, and thus different portions of the hash table will be constantly being transferred in and out of the cache, resulting to poor performance.

The above observations motivated us to explore whether changing the order in which packets are delivered from the OS to the monitoring application improves packet processing performance. Specifically, we speculated that rearranging the captured traffic stream so that packets of the same class are delivered to the application in "batches" would improve the locality of code and data accesses, and thus reduce the overall cache miss ratio. This rearrangement can be conceptually achieved by buffering arriving packets into separate "buckets," one for each packet class, and dispatching each bucket at once, either whenever it gets full, or after some predefined timeout since the arrival of the first packet in the bucket. For instance, if we assume that packets with the same destination port number correspond to the same class, then interleaved packets destined to different network services will be rearranged so that packets destined to the same network service are delivered back-to-back to the monitoring application, as depicted in Figure 1.

Choosing the destination port number as a class identifier strikes a good balance between the number of required buckets and the achieved locality for commonly used network monitoring applications. Indeed, choosing a more fine-grained classification scheme, such as a combination of the destination IP address and port number, would require a tremendous amount of buckets, and would probably just add overhead, since most of the applications of interest to this work perform (5-tuple) flow-based classification. At the same time, packets destined to the same port usually correspond to the same application-level protocol, so they will trigger the same Snort signatures and preprocessors, or will belong to the same or "neighboring" entries in a network flow hash table.

However, sorting the packets by destination port only would completely separate the two directions of each bi-directional flow, *i.e.*, client requests from server responses. This would increase significantly the distance between request and response packets, and in case of TCP flows, the distance between SYN and a SYN/ACK packets. For traffic processing operations that require to inspect both directions of a connection, this would add a significant delay, and eventually decrease memory locality, due to the separation of each bi-directional flow in two parts. Moreover, TCP reassembly would suffer from extreme buffering until the reception of pending ACK packets, or even discard the entire flow. For example, this could happen in case that ACKs are not received within a timeout period, or a packet is received before the SYN packet, *i.e.*, before the TCP connection establishment. Furthermore, splitting the two directions of a flow would alter the order in which the packets are delivered to the application. This could cause problems to applications that expect the captured packets to be delivered with monotonically increasing timestamps.

Based on the above, we need a sorting scheme that will be able to keep the packets of both directions of a flow together, in the same order, and at the same time maintain the benefits of packet sorting based on destination port: good locality and lightweight implementation. Our choice is based on the observation that the server port number, which commonly characterizes the class of the flow, is usually lower than the client port number, which is usually a high port number randomly chosen by the OS. Also, both directions of a flow have the same pair of port numbers, in just reverse order. Packets in server-to-client direction have the server's port as source port number. Hence, in most cases, choosing the smaller port between the source and destination port numbers of each packet will give us the server's port in both directions. In case of known services, low ports are almost always used. In case of peer-to-peer traffic or other applications that may use high server-side port numbers, connections between peers are established using high ports only. However, sorting based on any of these two ports has the same effect to the locality of the application's memory accesses. Sorting always based on the smaller among the two port numbers ensures that packets from both directions will be clustered together, and their relative order will always be maintained. Thus, our choice is to sort the packets according to the smaller between the source and destination ports.

Performance metric	Original trace	Sorted trace	Improvement
Throughput (Mbit/s)	473.97	596.15	25.78%
Cache misses (per packet)	11.06	1.33	87.98%
CPU cycles (per packet)	31,418.91	24,657.98	21.52%

Table 1: Snort's performance using a sorted trace.

3.1. Feasibility Estimation

To get an estimation of the feasibility and the magnitude of improvement that locality buffering can offer, we performed a preliminary experiment whereby we sorted off-line the packets of a network trace based on the lowest between the source and destination port numbers, and fed it to a passive monitoring application. This corresponds to applying locality buffering using buckets of infinite size. Details about the trace and the experimental environment are discussed in Section 5. We ran Snort v2.9 [1] using both the sorted and the original trace, and measured the processing throughput (trace size divided by user time), L2 cache misses, and CPU cycles of the application. Snort was configured with all the default preprocessors enabled as specified in its default configuration file and used the latest official rule set [21] containing 19,009 rules. The Aho-Corasick algorithm was used for pattern matching [22]. The L2 cache misses and CPU clock cycles were measured using the PAPI library [23], which utilizes the hardware performance counters.

Table 3.1 summarizes the results of this experiment (each measurement was repeated 100 times, and we report the average values). We observe that sorting results to a significant improvement of more than 25% in Snort's packet processing throughput, L2 cache misses are reduced by more than 8 times, and 21% less CPU cycles are consumed.

From the above experiment, we see that there is a significant potential of improvement in packet processing throughput using locality buffering. However, in practice, rearranging the packets of a continuous packet stream can only be done in short intervals, since we cannot indefinitely wait to gather an arbitrarily large number of packets of the same class before delivering them to the monitoring application—the captured packets have to be eventually delivered to the application within a short time interval (in our implementation, in the orders of milliseconds). Note that slightly relaxing the in-order delivery of the captured packets results to a delay between capturing the

packet, and actually delivering it to the monitoring application. However, such a sub-second delay does not actually affect the correct operation of the monitoring applications that we consider in this work (delivering an alert or reporting a flow record a few milliseconds later is totally acceptable). Furthermore, packet timestamps are computed *before* locality buffering, and are not altered in any way, so any inter-packet time dependencies remain intact.

4. Implementation within `libpcap`

We have chosen to implement locality buffering within `libpcap`, the most widely used packet capturing library, which is the basis for a multitude of passive monitoring applications. Typically, applications read the captured packets through a call such as `pcap_next` or `pcap_loop`, one at a time, in the same order as they arrive to the network interface. By incorporating locality buffering within `libpcap`, monitoring applications continue to operate as before, taking advantage of locality buffering in a transparent way, without the need to alter their code or link them with extra libraries. Indeed, the only difference is that consecutive calls to `pcap_next` or similar functions will most of the time return packets with the same destination or source port number, depending on the availability and the time constraints, instead of highly interleaved packets with different port numbers.

4.1. Periodic Packet Stream Sorting

In `libpcap`, whenever the application attempts to read a new packet, *e.g.*, through a call to `pcap_next`, the library reads a packet from kernel and delivers it to the application. Using `pcap_loop`, the application registers a callback function for packet processing that is called once per each captured packet read from kernel by `libpcap`. In case that memory mapping is not supported, the packet is copied through a `recv` call from kernel space to user space in a small buffer equal to the maximum packet size, and then `pcap_next` returns a pointer to the beginning of the new packet or the callback function registered by `pcap_loop` is called. With memory mapping, the next packet stored by kernel in the shared ring buffer is returned to application or processed by the callback function. If no packets are stored, `poll` is called to wait for the next packet reception.

So far, we have conceptually described locality buffering as a set of buckets, with packets having the same source or destination port ending up into the same bucket. One straightforward implementation of this approach would

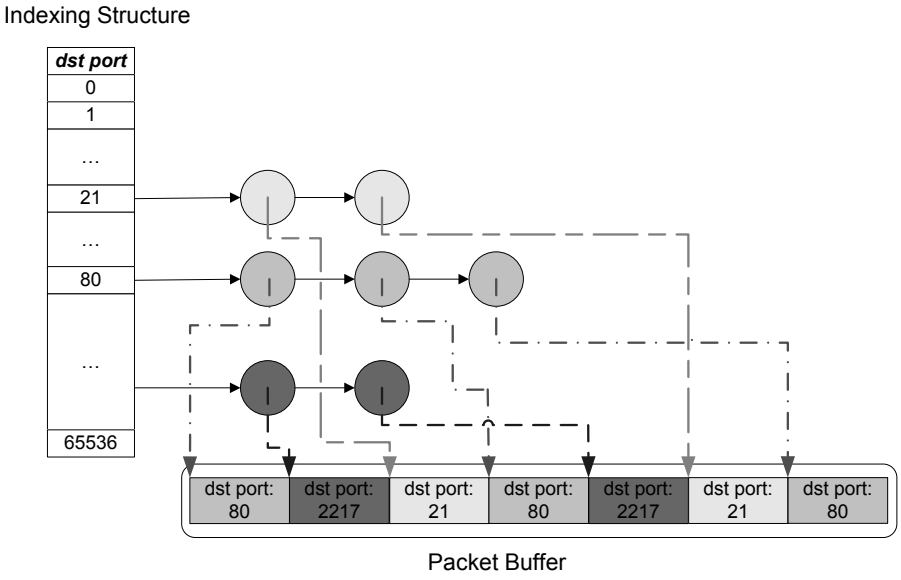

Figure 2: Using an indexing table with a linked list for each port, the packets are delivered to the application sorted by their smaller port number.

be to actually maintain a separate buffer for each bucket, and copy each arriving packet to its corresponding buffer. However, this has the drawback that an extra copy is required for storing each packet to the corresponding bucket, right after it has been fetched from the kernel.

In order to avoid additional packet copy operations, which incur significant overhead, we have chosen an alternative approach. We distinguish between two different phases: the packet *gathering* phase, and the packet *delivery* phase. In the case without memory mapping, we have modified the single-packet-sized buffer of `libpcap` to hold a large number of packets instead of just one. During the packet gathering phase, newly arrived packets are written sequentially into the buffer by increasing the buffer offset in the `recv` call until the buffer is full or a certain timeout has expired. For `libpcap` implementation with memory mapping support, the shared buffer is split into two parts. The first part of the buffer is used for gathering packets in the gathering phase, and the second part for delivering packets based on the imposed sorting. The gathering phase lasts either till the buffer used for packet gathering gets full or till a timeout period expires.

Instead of arranging the packets into different buckets, which requires an

extra copy operation for each packet, we maintain an index structure that specifies the order in which the packets in the buffer will be delivered to the application during the delivering phase, as illustrated in Figure 2. The index consists of a table with 64K entries, one for each port number. Each entry in the table points to the beginning of a linked list that holds references to all packets within the buffer with the particular port number. In the packet delivery phase, the packets are delivered to the application ordered according to their smaller port by traversing each list sequentially, starting from the first non-empty port number entry. In this way we achieve the desired packet sorting, while, at the same time, all packets remain in place, at the initial memory location in which they were written, avoiding extra costly copy operations. In the following, we discuss the two phases in more detail.

In the beginning of each packet gathering phase the indexing table is zeroed using `memset()`. For each arriving packet, we perform a simple protocol decoding for determining whether it is a TCP or UDP packet, and consequently extract its source and destination port numbers. Then, a new reference for the packet is added to the corresponding linked list. For non-TCP or non-UDP packets, a reference is added into a separate list. The information that we keep for every packet in each node of the linked lists includes the packet's length, the precise timestamp of the time when the packet was captured, and a pointer to the actual packet data in the buffer.

Instead of dynamically allocating memory for new nodes in the linked lists, which would be an overkill, we pre-allocate a large enough number of spare nodes, equal to the maximum number of packets that can be stored in the buffer. Whenever a new reference has to be added in a linked list, a spare node is picked. Also, for fast insertion of new nodes at the end of the list, we keep a table with 64K pointers to the tail of each list.

The overhead of this indexing process is negligible. We measured it using a simple analysis application, which just receives packets in user space and then discards them, resulting to less than 6% overhead for any traffic rate. This is because most of the CPU time in this application is spent for capturing packets and delivering them to user space. The overhead of finding the port numbers and adding a node to our data structure for each packet is negligible compared to packet capturing and other per-packet overheads. The overhead of making zero (through a `memset()` call) the indexing table is also negligible, since we make it once for a large group of packets. In this measurement we used a simple analysis application which does not benefit from improved

cache memory locality. For real world applications this overhead is even smaller, and as we observe in our experimental evaluation (Section 5) the benefits from memory locality enhancements outreach by far this overhead.

The system continues to gather packets until the buffer becomes full or a certain timeout has elapsed. The timeout ensures that if packets arrive with a low rate, the application will not wait too long for receiving the next batch of packets. The buffer size and the timeout are two significant parameters of our approach, since they influence the number of sorted packets that can be delivered to the application in each batch. Both timeout and buffer size can be defined by the application. Depending on the per-packet processing complexity of each application, the buffer size determines the benefit in its performance. In Section 5 we examine the effect that the number of packets in each batch has on the overall performance using three different passive monitoring applications. The timeout parameter is mostly related to the network's workload.

Upon the end of the packet gathering phase, packets can be delivered to the application following the order imposed by the indexing structure. For that purpose, we keep a pointer to the list node of the most recently delivered packet. Starting from the beginning of the index table, whenever the application requests a new packet, *e.g.*, through `pcap_next`, we return the packet pointed either by the next node in the list, or, if we have reached the end of the list, by the first node of the next non-empty list. The latter happens when all the packets of the same port have been delivered (*i.e.*, the bucket has been emptied), so conceptually the system continues with the next non-empty group.

4.2. Using a Separate Thread for Packet Gathering

In case that memory mapping is not supported in the system, a single buffer will be used for both packet gathering and delivery. A drawback of the above implementation is that during the packet gathering phase, the CPU remains idle most of the time, since no packets are delivered to the application for processing in the meanwhile. Reversely, during the processing of the packets that were captured in the previous packet gathering period, no packets are stored in the buffer. In case that the kernel's socket buffer is small and the processing time for the current batch of packets is increased, it is possible that a significant number of packets may get lost by the application in case of high traffic load.

Although in practice this effect does not degrade performance when short timeouts are used, we can improve further the performance of locality buffering in this case by employing a separate thread for the packet gathering phase, combined with the usage of two buffers instead of a single one. The separate packet gathering thread receives the packets from the kernel and stores them to the *write buffer*, and also updates its index. In parallel, the application receives packets for processing from the main thread of `libpcap`, which returns the already sorted packets of the second *read buffer*. Each buffer has its own indexing table.

Upon the completion of both the packet gathering phase, *i.e.*, after the timeout expires or when the write buffer becomes full, and the parallel packet delivery phase, the two buffers are swapped. The write buffer, which now is full of packets, turns to a read buffer, while the now empty read buffer becomes a write buffer. The whole swapping process is as simple as swapping two pointers, while semaphore operations ensure the thread-safe exchange of the two buffers.

4.3. Combine Locality Buffering and Memory Mapping

A step beyond is to combine locality buffering with memory mapping to further increase the performance of each individual technique. While memory mapping improves the performance of packet capturing, locality buffering aims to improve the performance of the user application that processes the captured packets.

As we described in Section 2.2, the buffer where the network packets are stored in `libpcap` with memory mapping support is accessible from both the kernel and `libpcap` library. The packets are stored sequentially into this buffer by the kernel as they arrive, while the `libpcap` library allows a monitoring application to process them by returning a pointer to the next packet through `pcap_next` or calling the callback function registered through `pcap_loop` for each packet that arrives.

In case the buffer is empty, `libpcap` blocks, calling `poll`, waiting for new packets to arrive.

After finishing with the processing of each packet, through the callback function or when the next `pcap_next` is called, `libpcap` marks the packet as read so that the kernel can later overwrite the packet with a new one. Otherwise, if a packet is marked as unread, the kernel is not allowed to copy a new packet into this position of the buffer. In this way, any possible data

corruption that could happen by the parallel execution of the two processes (kernel and monitoring application) is avoided.

The implementation of locality buffering in the memory mapped version of `libpcap` does not require to maintain a separate buffer for sorting the arriving packets, since we have direct access to the shared memory mapped buffer in which they are stored. To deliver the packets sorted based on the source or destination port number to the application, we process a small portion of the shared buffer each time as a batch: instead of executing the handler function every time a new packet is pushed into the buffer, we wait until a certain amount of packets has been gathered or a certain amount of time has been elapsed. The batch of packets is then ordered based on the smaller of source and destination port numbers.

The sorting of the packets is performed as described in Section 4.1. The same indexing structure, as depicted in Figure 2, was built to support the sorting. The structure contains pointers directly to the packets on the shared buffer. Then, the handler function is applied iteratively on each indexed packet based on the order imposed by the indexing structure. After the completion of the handler function, the packet is marked for deletion as before in order to avoid any race conditions between the kernel process and the user-level library.

A possible weakness of not using an extra buffer, as described in Section 4.2, is that if the batch of the packets is large in comparison to the shared buffer, a significant number of packets may get lost during the sorting phase in case of high traffic load. However, as discussed in Section 5, the fraction of the packets that we need to sort is very small compared to the size of the shared buffer. Therefore, it does not affect the insertion of new packets in the meanwhile.

In case of memory mapping, a separate thread for the packet gathering phase is not required. New incoming packets are captured and stored into the shared buffer by the kernel in parallel with the packet delivery and processing phase, since kernel and user level application (including the `libpcap` library) are two different processes. Packets that have been previously stored in buffer by kernel are sorted in batches during the gathering phase and then each sorted batch of packets are delivered one-by-one to the application for further processing.

5. Experimental Evaluation

5.1. Experimental Environment

Our experimental environment consists of two PCs interconnected through a 10 Gbit switch. The first PC is used for traffic generation, which is achieved by replaying real network traffic traces at different rates using `tcpreplay` [24]. The traffic generation PC is equipped with two dual-core Intel Xeon 2.66 GHz CPU with 4 MB L2 cache, 6 GB RAM, and a 10 Gbit network interface (SMC 10G adapter with XFP). This setup allowed us to replay traffic traces with speeds up to 2 Gbit/s. Achieving larger speeds was not possible using large network traces because usually the trace could not be effectively cached in main memory.

By rewriting the source and destination MAC addresses in all packets, the generated traffic is sent to the second PC, the passive monitoring sensor, which captures the traffic and processes it using different monitoring applications. The passive monitoring sensor is equipped with two quad-core Intel Xeon 2.00 GHz CPUs with 6 MB L2 cache, 6 GB RAM, and a 10 Gbit network interface (SMC 10G adapter with XFP). The size of memory mapped buffer was set to 60,000 frames in all cases, in order to minimize packet drops due to short packet bursts. Indeed, we observe that when packets are dropped by kernel, in higher traffic rates, the CPU utilization in the passive monitoring sensor is always 100%. Thus, in our experiments, packets are lost due to the high CPU load. Both PCs run 64bit Ubuntu Linux (kernel version 2.6.32).

For the evaluation we use a one-hour full payload trace captured at the access link that connects an educational network with thousands of hosts to the Internet. The trace contains 58,714,906 packets, corresponding to 1,493,032 different flows, totalling more than 40 GB in size. To achieve high speeds, up tp 2 Gbit/s, we split the trace into a few smaller parts, which can be effectively cached in the 6 GB main memory, and we replay each part of the trace for 10 times in each experiment.

We measure the performance of the monitoring applications on top of the original version of `libpcap-1.1.1` and our mofidied version with locality buffering. The latter combines locality buffering with the memory mapping. For each setting, we measure the L2 cache misses and the CPU clock cycles by reading the CPU performance counters through the PAPI library [23]. Another important metric we measure is the percentage of packets being

dropped by `libpcap`, which is occurred when replaying traffic in high rates due to high CPU utilization.

Traffic generation begins after the application has been initiated. The application is terminated immediately after capturing the last packet of the replayed trace. All measurements were repeated 10 times and we report the average values. We focus mostly on the discussion of our experiments using Snort IDS, which is the most resource-intensive among the tested applications. However, we also briefly report on our experiences with Appmon and Fprobe monitoring applications.

5.2. Snort

As in the experiments of Section 3.1, we ran Snort v2.9 using its default configuration, in which all the default preprocessors were enabled, and we used the latest official rule set [21] containing 19,009 rules. Initially, we examine the effect that the size of the buffer in which the packets are sorted has on the overall application performance. We vary the size of the buffer from 100 to 32,000 packets while replaying the network trace at a constant rate of 250 Mbit/s. We send traffic at several rates, but we first present results from constant 250 Mbit/s since no packets were dropped at this rate, to examine the effect of buffer size on CPU cycles spent and L2 cache misses when no packets are lost. We do not use any timeout in these experiments for packet gathering. As long as we send traffic at constant rate, the buffer size determines how long the packet gathering phase will last. Respectively, a timeout value corresponds to a specific buffer size.

Figures 3 and 4 show the per-packet CPU cycles and L2 cache misses respectively when Snort processes the replayed traffic using the original and modified versions of `libpcap`. Both `libpcap` versions use the memory mapping support, with the same size for the shared packet buffer (60,000 frames) for fairness. Figure 5 presents the percentage of the packets that are being dropped by Snort when replaying the traffic at 2 Gbit/s, for each different version of `libpcap`.

We observe that increasing the size of the buffer results to fewer cache misses, fewer clock cycles, less dropped packets, and generally to an overall performance improvement for the locality buffering implementations. This is because using a larger packet buffer offers better possibilities for effective packet sorting, and thus to better memory locality. However, increasing the size from 8,000 to 32,000 packets gives only a slight improvement. Based on this result, we consider 8,000 packets as an optimum buffer size in our

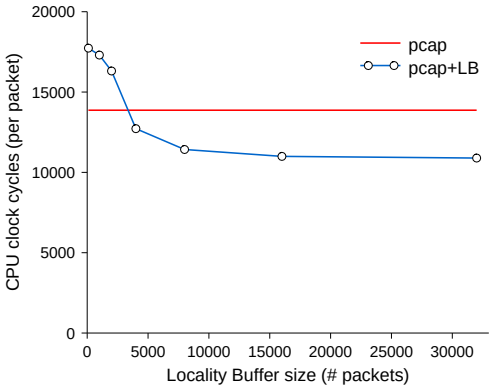

Figure 3: Snort's CPU cycles as a function of the buffer size for 250 Mbit/s traffic.

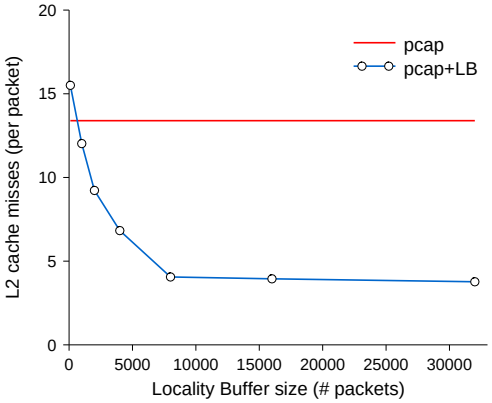

Figure 4: Snort's L2 cache misses as a function of the buffer size for 250 Mbit/s traffic.

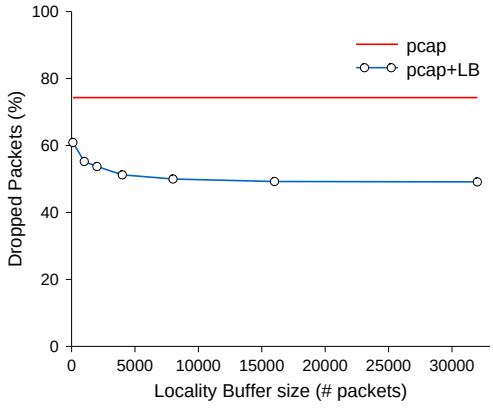

Figure 5: Snort's packet loss ratio as a function of the buffer size for 2 Gbit/s traffic.

experiments. When sending traffic in a constant rate of 250 Mbit/s, with no timeout specified, 8,000 packets as buffer size roughly correspond to an 128 millisecond period at average.

We can also notice that using locality buffering we achieve a significant reduction in L2 cache misses from 13.4 per packet to 4.1, when using a 8,000-packet buffer, which is an improvement of 3.3 times against Snort with the original `libpcap` library. Therefore, Snort's user time and clock cycles are significantly reduced using locality buffering, making it faster by more than 20%. Moreover, when replaying the traffic at 2 Gbit/s, the packet loss ratio is reduced by 33%. Thus, Snort with locality buffering and memory mapped

`libpcap` performs significantly better than using the original `libpcap` with memory mapping support. When replaying the trace at low traffic rates, with no packet loss, Snort outputs the same set of alerts with and without locality buffering, so the packet reordering does not affect the correct operation of Snort's detection process.

We repeated the experiment by replaying the trace in different rates ranging from 100 to 2,000 Mbit/s and in every case we observed a similar behavior. In all rates, 8,000 packets was found to be the optimum buffer size. Using this buffer size, locality buffering results in all rates to a significant reduction in Snort's cache misses and CPU cycles, similar to the improvement observed for 250 Mbit/s traffic against the original `libpcap`. The optimum buffer size depends mainly on the nature of traffic in the monitored network and on the network monitoring application's processing.

An important metric for evaluating the performance of our implementations is the percentage of the packets that are being dropped in high traffic rates by the kernel due to high CPU load, and the maximum processing throughput that Snort can sustain without dropping packets. In Figure 6 we plot the average percentage of packets that are being lost while replaying the trace with speeds ranging from 100 to 2,000 Mbit/s, with a step of 100 Mbit/s. The 2,000 Mbit/s limitation is due to caching the trace file parts from disk to main memory in the traffic generator machine, in order to generate real network traffic. We used a 8,000-packet locality buffer, which was found to be the optimal size for Snort when replaying our trace file at any rate.

Using the unmodified `libpcap` with memory mapping, Snort cannot process all packets in rates higher than 600 Mbit/s, so a significant percentage of packets is being lost. On the other hand, using locality buffering the packet processing time is accelerated and the system is able to process more packets in the same time interval. As shown in Figure 6, using locality buffering Snort becomes much more resistant in packet loss, and starts to lose packets at 1 Gbit/s instead of 600 Mbit/s. Moreover, at 2 Gbit/s, our implementation drops 33% less packets than the original `libpcap`.

Figure 7 shows Snort's CPU utilization as a function of the traffic rate, for rates varying from 100 Mbit/s to 2 Gbit/s, with a 100 Mbit/s step, and a locality buffer size of 8,000 packets. We observe that for low speeds, locality buffering reduces the number of CPU cycles spent for packet processing due to the improved memory locality. For instance, for 500 Mbit/s, Snort's CPU utilization with locality buffering is reduced by 20%. The CPU utilization

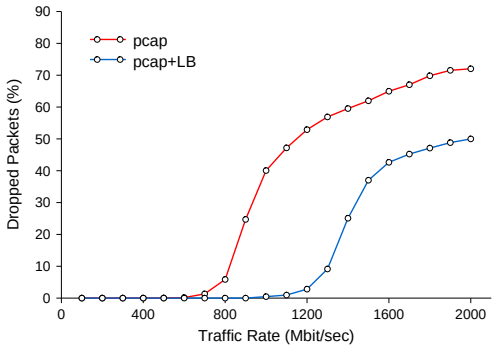

Figure 6: Packet loss ratio of the passive monitoring sensor when running Snort, as a function of the traffic speed for an 8,000-packet locality buffer.

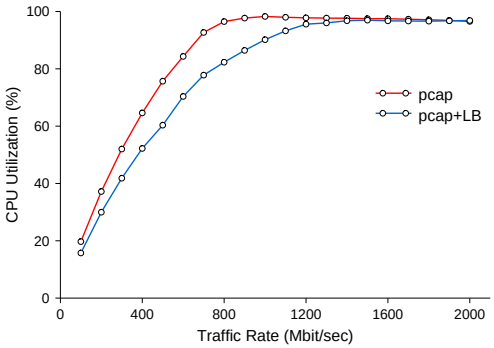

Figure 7: CPU utilization in the passive monitoring sensor when running Snort, as a function of the traffic speed for an 8,000-packet locality buffer.

when Snort does not use locality buffering exceeds 90% for rates higher than 600 Mbit/s, and reaches up to 98.3% for 1 Gbit/s. On the other hand, using locality buffering, Snort's CPU utilization exceeds 90% for rates higher than 1 Gbit/s, and reaches about 97% for 1.5 Gbit/s rate. We also observe that packet loss events occur due to high CPU utilization, when it approaches 100%. Without locality buffering, 92.7% CPU utilization for 700 Mbit/s results to 1.4% packet loss rate, while 98% utilization for 1.1 Gbit/s results to 47.2% packet loss.

In Figure 8 we plot the average percentage of dropped packets while replaying traffic with normal timing behavior, instead of sending traffic with a constant rate. We replay the traffic of our trace based on it's normal traffic patterns when the trace was captured, using the multiplier option of tcpreplay [24] tool. Thus, we are able to replay the trace at the speed that it was recorded, which is 88 Mbit/s on average, or at a multiple of this speed. In this experiment we examine the performance of Snort using the original and our modified `libpcap` in case of normal traffic patterns with packet bursts, instead of constant traffic rates. We send the trace using multiples from 1 up to 16, and we plot the percentage of dropped packets as a function of this multiplication factor. We use 8,000 packets as buffer size and 100 ms timeout.

We observe that locality buffering reduces the percentage of dropped packets in higher traffic rates, when using larger multiplication factors. Snort with locality buffering starts dropping packets when sending traffic eight

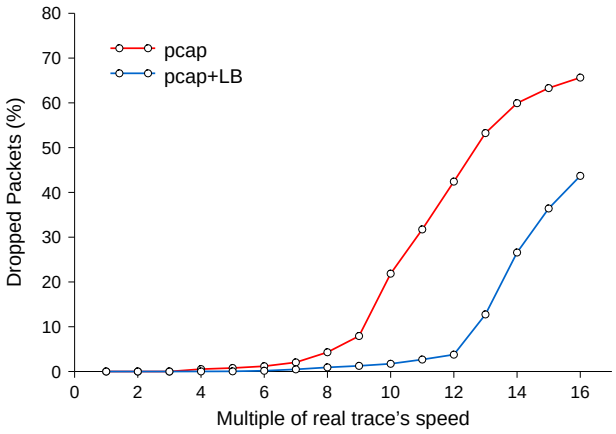

Figure 8: Packet loss ratio of the passive monitoring sensor when running Snort, as a function of the actual trace's speed multiple, with an 8,000-packet locality buffer and 100 ms timeout.

times faster than the actual speed of the trace, while Snort with the original `libpcap` drops packets from four times faster speed. When replaying traffic 16 times faster than the recorded speed, which results to 1,408 Mbit/s on average, Snort with locality buffering drops 34% less packets.

Since both versions of `libpcap` use a ring buffer for storing packets with the same size, they are resistant to packet bursts at a similar factor. However, `libpcap` with locality buffering is faster, due the improved memory locality, and so it is more resistant to packet drops in cases of overloads and traffic bursts.

5.3. Appmon

`Appmon` [2] is a passive network monitoring application for accurate per-application traffic identification and categorization. It uses deep-packet inspection and packet filtering for attributing flows to the applications that generate them. We ran Appmon on top of our modified version of `libpcap` to examine its performance using different buffer sizes that vary from 100 to 32,000 packets, and compare with the original `libpcap`. Figure 9 presents the Appmon's CPU cycles and Figure 10 the L2 cache misses measured while replaying the trace at a constant rate of 500 Mbit/s. At this rate no packet loss was occurred.

The results show that Appmon's performance can be improved using the locality buffering implementation, reducing the CPU cycles by about

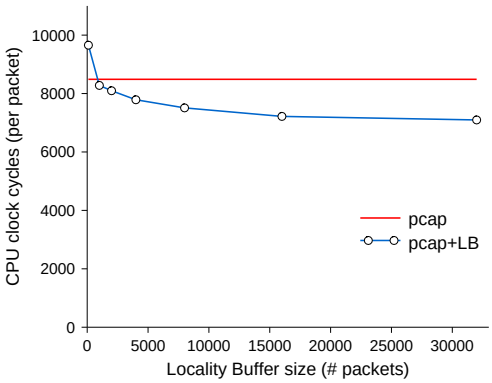

Figure 9: Appmon's CPU cycles as a function of the buffer size for 500 Mbit/s traffic.

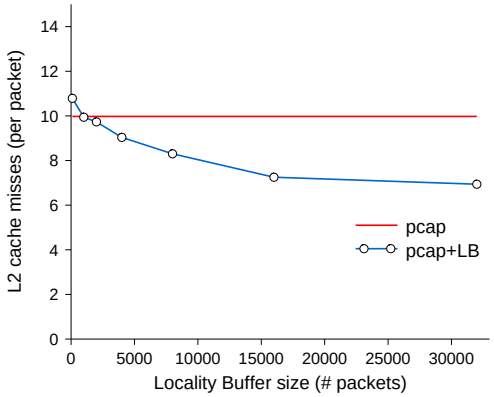

Figure 10: Appmon's L2 cache misses as a function of the buffer size for 500 Mbit/s traffic.

16% compared to Appmon using the original `libpcap`. Cache misses are reduced by up to 31% for 32,000 packets buffer size and 28% for 16,000 packets. We notice that in case of Appmon the optimum buffer size is around 16,000 packets, while in Snort 8,000 packets size is enough to optimize the performance. This happens because Appmon is not so CPU-intensive as Snort, so it requires a larger amount of packets to be sorted in order to achieve a significant performance improvement.

We ran Appmon with traffic rates varying from 250 to 2,000 Mbit/s, observing similar results. Since Appmon does less processing than snort, less packets are dropped in high rates. The output of Appmon remains identical in all cases, which means that the periodic packet stream sorting does not affect the correct operation of Appmon's classification process.

5.4. Fprobe

`Fprobe` [3] is a passive monitoring application that collects traffic statistics for each active flow and exports the corresponding NetFlow records. We ran Fprobe with the original and our modified version of `libpcap` and performed the same measurements as with Appmon.

Figure 11 plots the CPU cycles and Figure 12 the L2 cache misses of the Fprobe variants for buffer sizes from 100 up to 32,000 packets, while replaying the trace at a rate of 500 Mbit/s.

We notice a speedup of about 11% in Fprobe when locality buffering is enabled, for 4,000 packets buffer size, while cache misses are reduced by

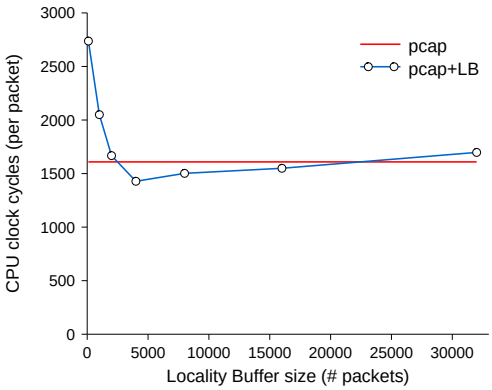

Figure 11: Fprobe's CPU cycles as a function of the buffer size for 500 Mbit/s traffic.

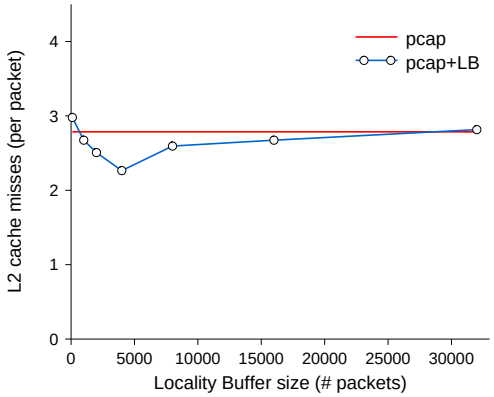

Figure 12: Fprobe's L2 cache misses as a function of the buffer size for 500 Mbit/s traffic.

19%. The buffer size that optimizes the overall performance of Fprobe in this setup is around 4,000 packets. No packet loss occurred for Fprobe at all traffic rates. Fprobe is even less CPU-intensive than Appmon and Snort since it performs only a few operations per packet. The time spent in kernel for packet capturing is significantly larger than the time spent in user space. Thus, Fprobe benefits less from the locality buffering enhancements. For passive monitoring applications in general, the performance improvement due to locality buffering increases as the time spent in user space increases, and also depends on memory access patterns. Similar results were observed for all rates of the replayed traffic.

6. Related Work

The concept of locality buffering for improving passive network monitoring applications, and, in particular, intrusion detection and prevention systems, was first introduced by Xinidis *et al.* [25], as part of a load balancing traffic splitter for multiple network intrusion detection sensors that operate in parallel. In this work, the load balancer splits the traffic to multiple intrusion detection sensors, so that similar packets (*e.g.*, packets destined to the same port) are processed by the same sensor. However, in this approach the splitter uses a limited number of locality buffers and copies each packet to the appropriate buffer based on hashing on its destination port number. Our approach differs in two major aspects. First, we have implemented locality

buffering within a packet capturing library, instead of a separate network element. To the best of our knowledge, our prototype implementation within the libpcap library is the first attempt for providing memory locality enhancements for accelerating packet processing in a generic and transparent way for existing passive monitoring applications. Second, the major improvement of our approach is that packets are not actually copied into separate locality buffers. Instead, we maintain a separate index which allows for scaling the number of locality buffers up to 64K.

Locality enhancing techniques for improving server performance have been widely studied. For instance, Markatos *et al.* [26] present techniques for improving request locality on a Web cache, which results to significant improvements in the file system performance.

Braun *et al.* [15] and Schneider *et al.* [27] compare the performance of packet capturing libraries on different operating systems using the same hardware platforms and provide guidelines for system configuration to achieve optimal performance. Several research efforts [12, 13, 14, 17] have focused on improving the performance of packet capturing through kernel and library modifications. These approaches reduce the time spent in kernel and the number of memory copies required for delivering each packet to the application. In contrast, our locality buffering approach aims to improve the packet processing performance of the monitoring application itself, by exploiting the inherent locality of the in-memory workload of the application. Moreover, such approaches can be integrated with our locality buffering technique to achieve the maximum possible improvement in CPU-intensive passive monitoring applications running in commodity hardware.

Schneider *et al.* [16] show that commodity hardware and software may be able to capture traffic for rates up to 1 Gbit/s, mainly due to limitations with buses bandwidth and CPU load. To cope with this limitation, the authors propose a monitoring architecture for higher speed interfaces by splitting the traffic across a set of nodes with lower speed interfaces, using a feature of current Ethernet switches. The detailed configuration for load balancing is left for the application. Vallentin *et al.* [28] present a NIDS cluster based on commodity PCs. Some front-end nodes are responsible to distribute the traffic across the cluster's back-end nodes. Several traffic distribution schemes are discussed, focused on minimizing the communication between the sensors and keeping them simple enough to be implemented effectively in the front-end nodes. Hashing a flow identifier is proposed as the right choice. A locality-aware approach in such architectures would cluster similar flows to

the same node, by sorting and splitting packets based on source and destination port numbers, to improve memory locality and cache usage in each individual node.

Fusco and Deri [29] utilize the RSS feature of modern NICs [30], which split the network traffic in multiple RX queues, usually equal to the number of CPU cores, to parallelize packet capturing using all CPU cores. Moreover, packets are copied directly from each hardware queue to a corresponding ring buffer, which is exposed in user-level as a virtual network interface. Thus, applications can easily and efficiently split the load to multiple threads or processes without contention. The load distribution in the several queues is based on a hash function, so it's not optimized in terms of memory locality and L2 cache usage. A locality-aware load distribution algorithm, like our approach, would split packets in queues based on their source and destination port numbers. Thus, both directions of each flow as well as similar flows would fall to the same queue and would be processed by the same thread and same CPU core. Locality-aware and flow-based load balancing approaches could be combined to handle cases where most of the flows have a common port number, so that these flows can be splitted to more than one CPU cores.

Sommer *et al.* [31] utilize multicore processors to pararrelize event-based network prevention systems, using multiple event queues that collect together semantically related events for in-order execution. Since the events are related, keeping them within a single queue localizes memory access to shared state by the same thread. On the other hand, our approach clusters together similar packets, *i.e.*, packets belonging to the same or similar flow, with the same or close port numbers.

7. Discussion

We consider two main limitations of our locality buffering approach. The first limitation is that the packet reordering we impose results in non-monotonic increasing timestamps among different flows (it guarantees monotonic increasing timestamps only *per each* bi-directional flow). Therefore, applications that require this property, *e.g.*, for connection timeout issues, may have problems when dealing with non-monotonic increasing timestamps. Such applications should either be modified to handle this issue, otherwise they cannot use our approach.

The second limitation of our approach is that our generic implementation within `libpcap`, which sorts packets based on source or destination port

numbers, may not be suitable for applications that require a custom packet sorting or scheduling approach, *e.g.*, based on application's semantics. Monitoring applications may perform similar processing for packets with specific port numbers, which cannot be known to the packet capturing library. Such applications should not use our modified `libpcap` version, but instead implement a custom scheduling scheme for packet processing order. Our implementation's goal, is to improve *transparently* the performance of a large class of existing applications, where the packet processing tasks depend mainly on the packets port numbers.

In particular, locality buffering technique is intended for applications which perform similar processing and similar memory accesses for the same class of packets, *e.g.*, for packets of the same flow or packets belonging to the same higher level protocol or application. For instance, signature-based intrusion detection systems can benefit from locality buffering, due to different set of signatures matched against different classes of packets. Other types of monitoring applications may not gain the same performance improvement from locality buffering.

Finally, recent trends impose the use of multiple CPU cores per processor, instead of building and using faster processors. Thus, applications should utilize all the available CPU cores to take full advantage of modern hardware and improve their performance. Although L2 cache memory becomes larger in newest processors, more processor cores tend to access the shared L2 cache, so locality enhancements can still benefit the overall performance. Our approach can be extended to exploit memory locality enhancements for improving the performance of multithreaded applications running in multicore processors. Improving memory locality for each thread, which usually runs on a single core, is an important factor that can significantly improve packet processing performance. Each thread should process similar packets to improve its memory access locality, similarly to our approach, which is intended for single-threaded applications.

Applications usually choose to split packets to multiple threads (one thread per core) based on flow identifiers. A generic locality-aware approach for efficient packet splitting to multiple threads, in order to optimize cache usage in each CPU core, should sort packets based on their port numbers and then divide them to the multiple threads. This will lead to improved code and data locality in each CPU core, similarly to our locality buffering approach. However, in some applications the best splitting of packets to multiple threads can be done only by the application itself, based on custom

application's semantics, *e.g.*, custom sets of ports with similar processing. In these cases, a generic library for improved memory locality cannot be used.

8. Conclusion

In this paper, we present a technique for improving the packet processing performance in a wide range of passive network monitoring applications by enhancing the locality of code and data accesses. Our approach is based on reordering the captured packets before delivering them to the monitoring application by grouping together packets with the same source or destination port number. This results to improved locality in application code and data accesses, and consequently to an overall increase in the packet processing throughput and to a significant decrease in the packet loss rate.

To maximize improvements in processing throughput, we combine locality buffering with memory mapping, an existing technique in `libpcap` that optimizes the performance of packet capturing. By mapping a buffer into shared memory, this technique reduces the time spent in context switching for delivering packets from kernel to user space.

We describe in detail the design and implementation of locality buffering within `libpcap`. Our experimental evaluation using three representative passive monitoring applications shows that all applications gain a significant performance improvement when using the locality buffering implementations, while the system can keep up with higher traffic speeds without dropping packets. Specifically, locality buffering results to a 25% increase in the processing throughput of the Snort IDS and allows it to process two times higher traffic rates without packet drops.

Using the original `libpcap` implementation, the Snort sensor starts to drop packets when the monitored traffic speed reaches 600 Mbit/s, while using locality buffering, packet loss is exhibited beyond 1 Gbit/s. Fprobe, a NetFlow export probe, and Appmon, an accurate traffic classification application, also exhibit significant throughput improvements, up to 12% and 18% respectively, although they do not perform as CPU-intensive processing as Snort.

Overall, we believe that implementing locality buffering within `libpcap` is an attractive performance optimization, since it offers significant performance improvements to a wide range of passive monitoring applications, while at the same time its operation is completely transparent, without the need to modify existing applications. Our implementation of locality buffering in the

memory mapped version of `libpcap` offers even better performance, since it combines optimizations in both the packet capturing and packet processing phases.

Acknowledgments

This work was supported in part by the FP7-PEOPLE-2009-IOF project MALCODE and the FP7 project SysSec, funded by the European Commission under Grant Agreements No. 254116 and No. 257007. We thank the anonymous reviewers for their valuable feedback on earlier versions of this paper. Antonis Papadogiannakis, Giorgos Vasiliadis, Demetres Antoniades, and Evangelos Markatos are also with the University of Crete.

References

[1] M. Roesch, Snort: Lightweight intrusion detection for networks, in: Proceedings of the USENIX Systems Administration Conference (LISA), 1999.

[2] D. Antoniades, M. Polychronakis, S. Antonatos, E. P. Markatos, S. Ubik, A. Oslebo, Appmon: An application for accurate per application traffic characterization, in: Proceedings of the IST Broadband Europe Conference, 2006.

[3] Fprobe: Netflow probes, `http://fprobe.sourceforge.net/`.

[4] M. Grossglauser, J. Rexford, Passive traffic measurement for IP operations, in: The Internet as a Large-Scale Complex System, 2005, pp. 91–120.

[5] V. Paxson, Bro: A system for detecting network intruders in real-time, in: Proceedings of the 7th USENIX Security Symposium, 1998.

[6] M. Handley, V. Paxson, C. Kreibich, Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics, in: Proceedings of the 10th USENIX Security Symposium, 2001.

[7] S. McCanne, C. Leres, V. Jacobson, libpcap, lawrence Berkeley Laboratory, Berkeley, CA. (software available from `http://www.tcpdump.org/`).

[8] G. Insolvibile, Inside the linux packet filter, Linux Journal 94 (2002).

[9] J. C. Mogul, K. K. Ramakrishnan, Eliminating receive livelock in an interrupt-driven kernel, ACM Transactions on Computer Systems 15 (3) (1997) 217–252.

[10] J. H. Salim, R. Olsson, A. Kuznetsov, Beyond softnet, in: ALS '01: Proceedings of the 5th annual conference on Linux Showcase & Conference, 2001.

[11] S. McCanne, V. Jacobson, The BSD Packet Filter: A New Architecture for User-level Packet Capture, in: Proceedings of the Winter 1993 USENIX Conference, 1993, pp. 259–270.

[12] L. Deri, Improving passive packet capture:beyond device polling, in: Proceedings of the 4th International System Administration and Network Engineering Conference (SANE), 2004.

[13] L. Deri, ncap: Wire-speed packet capture and transmission, in: Proceedings of the IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services (E2EMON), 2005.

[14] A. Biswas, P. Sinha, On improving performance of network intrusion detection systems by efficient packet capturing, in: Proceedings of the 10th IEEE/IFIP Network Operations and Management Symposium (NOMS), Vancouver, Canada, 2006.

[15] L. Braun, A. Didebulidze, N. Kammenhuber, G. Carle, Comparing and improving current packet capturing solutions based on commodity hardware, in: Proceedings of the 10th annual conference on Internet Measurement (IMC), 2010, pp. 206–217.

[16] F. Schneider, J. Wallerich, A. Feldmann, Packet capture in 10-gigabit ethernet environments using contemporary commodity hardware, in: Proceedings of the 8th international conference on Passive and Active network Measurement (PAM), 2007, pp. 207–217.

[17] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, G. Portokalidis, FFPF: fairly fast packet filters, in: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation (OSDI), 2004.

[18] R. Watson, C. Peron, Zero-Copy BPF Buffers, FreeBSD Developer Summit.

[19] A. Fiveg, Ringmap Capturing Stack for High Performance Packet Capturing in FreeBSD, FreeBSD Developer Summit.

[20] Dag ethernet network monitoring cards, Endace measurement systems, `http://www.endace.com/`.

[21] Sourcefire vulnerability research team (vrt), `http://www.snort.org/vrt/`.

[22] A. V. Aho, M. J. Corasick, Efficient string matching: an aid to bibliographic search, Communications of the ACM 18 (1975) 333–340.

[23] Performance application programming interface, `http://icl.cs.utk.edu/papi/`.

[24] Tcpreplay, `http://tcpreplay.synfin.net/trac/`.

[25] K. Xinidis, I. Charitakis, S. Antonatos, K. G. Anagnostakis, E. P. Markatos, An active splitter architecture for intrusion detection and prevention, IEEE Transactions on Dependable and Secure Computing 03 (1) (2006) 31–44.

[26] E. P. Markatos, D. N. Pnevmatikatos, M. D. Flouris, M. G. H. Katevenis, Web-conscious storage management for web proxies, IEEE/ACM Transactions on Networking 10 (6) (2002) 735–748.

[27] F. Schneider, J. Wallerich, Performance evaluation of packet capturing systems for high-speed networks, in: Proceedings of the 2005 ACM Conference on Emerging Network Experiment and Technology (CoNEXT), 2005, pp. 284–285.

[28] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, B. Tierney, The NIDS cluster: Scalable, stateful network intrusion detection on commodity hardware, in: Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID), 2007, pp. 107–126.

[29] F. Fusco, L. Deri, High speed network traffic analysis with commodity multi-core systems, in: Proceedings of the 10th annual conference on Internet measurement (IMC), 2010, pp. 218–224.

[30] Intel Server Adapters, Receive side scaling on Intel Network Adapters, http://www.intel.com/support/network/adapter/pro100/sb/cs-027574.htm.

[31] R. Sommer, V. Paxson, N. Weaver, An architecture for exploiting multi-core processors to parallelize network intrusion prevention, Concurrency and Computation: Practice and Experience 21 (10) (2009) 1255–1279.