Improving the Performance of Passive Network Monitoring Applications using Locality Buffering

Antonis Papadogiannakis, Demetres Antoniades, Michalis Polychronakis, and Evangelos P. Markatos
Institute of Computer Science
Foundation for Research & Technology – Hellas
{papadog,danton,mikepo,markatos}@ics.forth.gr

***Abstract*—In this paper, we present a novel approach for improving the performance of a large class of CPU and memory intensive passive network monitoring applications, such as intrusion detection systems, traffic characterization applications, and NetFlow export probes. Our approach, called *locality buffering*, reorders the captured packets by clustering packets with the same destination port, before they are delivered to the monitoring application, resulting to improved code and data locality, and consequently to an overall increase in the packet processing throughput and to a decrease in the 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 the need to change application code. Our experimental evaluation shows that locality buffering improves significantly the performance of popular applications, such as the Snort IDS, which exhibits a 40% increase in the packet processing throughput and a 60% improvement in packet loss rate.**

I. Introduction

Passive network monitoring is the basis for a multitude of systems that support the robust, efficient, and secure operation of modern computer networks. 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 also 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]. The complex analysis operations of such demanding applications are translated into an increased number of CPU cycles spent on each captured packet, which 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 can keep up with higher traffic loads.

A common characteristic that is often found in such monitoring applications is that they usually perform different operations to different types of packets. For example, a NIDS applies a certain subset of attack signatures to packets with destination port 80, i.e., it applies the web-attack signatures to packets destined to web servers, it applies a different set of signatures to packets destined to database servers, and so on. Furthermore, NetFlow probes [3], traffic categorization, as well as TCP stream reassembly, which has become a mandatory function of modern NIDS, 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 found in many passive monitoring applications. In practice, the captured packet stream is a mix of interleaved packets corresponding to hundreds or thousands of different packet types, depending on the monitored link. Our approach, called *locality buffering*, is based on adapting the packet stream in a way that enhances the locality of the application's code and memory access, and thus accelerating overall packet processing performance.

We have implemented locality buffering in `libpcap` [4], 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 alter their code. The experimental evaluation of our prototype implementation with real-world applications shows that locality buffering can significantly improve packet processing throughput and reduce the packet loss rate. For instance, the popular Snort IDS exhibited a 40% increase in the packet processing throughput, and a 60% improvement in packet loss rate.

The rest of the paper is organized as follows: in Section II we describe the overall approach of locality buffering, while in Section III we present in detail our implementation of locality buffering within the `libpcap` packet capture library. Section IV presents the experimental evaluation of our prototype implementation using three popular passive monitoring tools. Finally, Section V summarizes related work and Section VI concludes the paper.

II. 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, while different packet classes result to the execution of different code paths and to data accesses to different memory locations. Such packet

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

TABLE I
SNORT'S PERFORMANCE USING A SORTED TRACE

Performance metric	Original trace	Sorted trace
Throughput (Mbit/sec)	188.39	286.18
Cache Misses (per packet)	18.86	2.79
Clock Cycles (per packet)	48,978.76	30,846.89

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 the 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 checking a newly arrived packet, the corresponding preprocessor(s) code, signature subset, and 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 classification 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 in such a way that packets of the same class are delivered to the application in "batches" would improve the locality of memory 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 emptying each bucket at once, either whenever it gets full, or after some predefined timeout since the arrival of the first packet of the bucket. For instance, if we assume that packets with the same destination port number correspond to the same class, then any interleaved packets destined to different network services will be rearranged so that packets to the same 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. 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 anyway. 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.

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 destination port number, 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 IV. We ran Snort [1] using both the sorted, as well as the original trace, and measured the processing throughput (trace size divided by the measured user plus system time), L2 cache misses, and CPU cycles of the application. Snort was configured with all the default preprocessors and signature sets enabled (2833 rules and 11 preprocessors). The L2 cache misses and CPU clock cycles were measured using the PAPI library [5], which utilizes the hardware performance counters.

Table I summarizes the results (each measurement was repeated 100 times, and we report the average values). We see that sorting results to a significant improvement of more than 50% in Snort's packet processing throughput, L2 cache misses are reduced by more than 6 times, and 40% 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.

III. 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`, one at a time, in the same order as they arrive to the network interface. By incorporating locality buffering withing `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 linking 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 port number, depending on the availability and the time constraints, instead of highly interleaved packets with different destination port numbers.

A. 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 the kernel through a `recv` call, and delivers it to the application. That is, the packet is copied 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.

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

In order to avoid extra 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. 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.

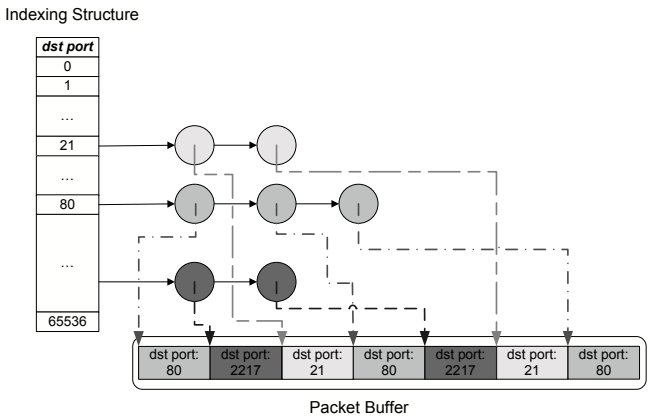

Fig. 2. Using an indexing table with a linked list for each port, the packets are delivered to the application sorted by their destination port.

Instead of arranging the packets into different buckets, which requires an extra copy operation for each packet, we maintain an indexing structure that specifies the order in which the packets in the buffer will be delivered to the application during the delivering phase. This indexing structure is illustrated in Figure 2. The index consists of a table with 64K entries, one for each port number. Each entry of the table points to the beginning of a linked list that holds references to all packets within the buffer with the particular destination port. In the packet delivery phase, traversing each list sequentially, starting from the first non-empty port number entry, allows for delivering the packets of the buffer ordered according to their destination port. In this way we achieve the desired packet sorting, while, at the same time, all packets remain in place, in the initial memory location where they had been written by `recv`, 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 destination port number. 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 to a linked list, a spare node is picked. Also, for fast insertion of new nodes at the end of the linked list, we keep a table with 64K pointers to the tail of each list.

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. We use 100ms as the default timeout in our prototype implementation, but both the timeout and the buffer size can be defined by the user. 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. Depending on how intensive each application is, this number of packets determines the benefit in its performance. In Section IV we examine the effect that the number of packets in each batch has on overall performance using three different passive monitoring applications.

Upon the end of the packet gathering phase, packets can be delivered to the application following the order imposed from 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 destination port have been delivered (i.e., the bucket has been emptied), so conceptually the system continues with the next non-empty group.

B. Using a Separate Thread for Packet Gathering

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 due to the very short timeouts used (e.g. 100ms), as we show in Section IV, we can improve further the performance of locality buffering 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.

IV. Experimental Evaluation

In this section, we present the experimental evaluation of our prototype implementation of locality buffering. Our experimental environment consists of two PCs interconnected through a Gigabit switch. The first PC is used for traffic generation, which is achieved by replaying real network traffic traces at different rates using `tcpreplay` [6]. We used a full payload trace captured at the access link that connects an educational network with thousands of hosts to the Internet. The trace contains 1,698,902 packets, corresponding to 64,628 different network flows, totaling more than 1 GB in size.

By rewriting the source and destination MAC addresses in all packets, the generated traffic can be 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 an Intel Xeon 2.40 GHz processor with 512 KB L2 cache and 512 MB RAM running Debian Linux (kernel version 2.6.18). The kernel socket buffer size was set to 16 MB, in order to minimize packet loss due to packet bursts.

We tested the performance of the monitoring applications on top of three different versions of `libpcap`: the original version, our modified version that employs locality buffering, and a third version with the optimized locality buffering approach that uses a separate thread for storing incoming packets. For each setting, we measured the application's user and system time using the UNIX `time` utility. Furthermore, we measured the L2 cache misses and the CPU clock cycles by reading the CPU performance counters through the PAPI library [5]. Finally, an important metric that was measured is the percentage of packets being dropped by `libpcap`, which usually happens when replaying the 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. Due to space constraints, we focus mostly on the discussion of our experiments using Snort, which is the most resource-intensive among the tested applications. However, we also briefly report our experiences with Fprobe and Appmon.

A. Snort

We ran Snort using its default configuration, in which almost all of the available rule sets and preprocessors are enabled. Snort loaded 2833 rules, while 11 preprocessors were active.

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 16000 packets while replaying the network trace at a constant rate of 100 Mbit/sec. Using a 100 Mbit/sec rate, no packets were dropped. We do not use any timeout in these experiments for packet gathering. As long as we send traffic at constant rate,

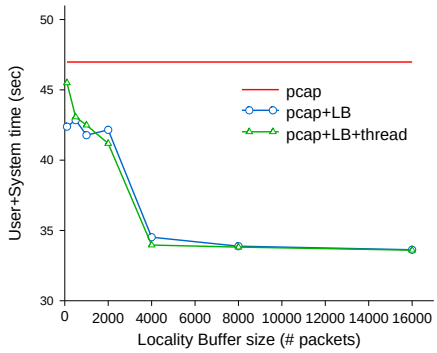

Fig. 3. Snort's user plus system time as a function of the buffer size for 100 Mbit/s traffic.

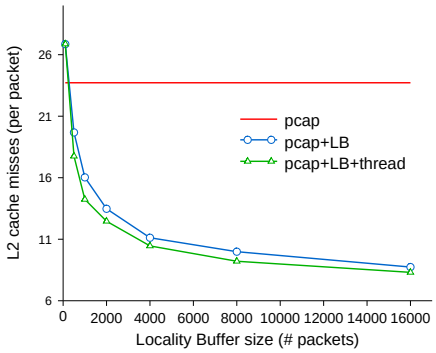

Fig. 4. Snort's L2 cache misses as a function of the buffer size for 100 Mbit/s traffic.

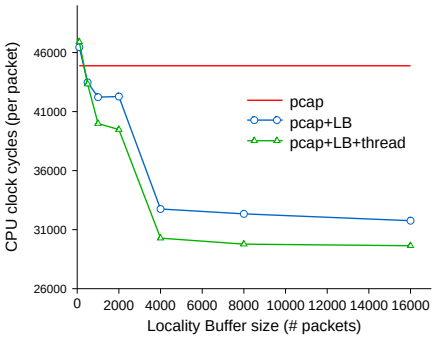

Fig. 5. Snort's CPU cycles as a function of the buffer size for 100 Mbit/s traffic.

the buffer size determines how long the packet gathering phase will last. Figure 3 shows the user plus system time of Snort for processing the replayed traffic using the different `libpcap` versions. Figures 4 and 5 present the per-packet L2 cache misses and clock cycles respectively.

We observe that increasing the size of the buffer results to lower user time, fewer cache misses and clock cycles, and generally to an overall performance improvement. 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 4000 to 16000 packets gives only a slight improvement. Based on this result, we consider 4000 packets as optimum buffer size in our experiments. For a rate of 100 Mbit/sec, 4000 packets roughly correspond to an 160 millisecond period at average.

We can also notice that using locality buffering we achieve a significant reduction on the L2 cache misses from 23.7 per packet to 10.5, when using a 4000 packets buffer, which is an improvement of 2.26 times against Snort with the original `libpcap` library. Also, Snort's user time and clock cycles are significantly reduced, making it faster by more than 40%.

Comparing our two different implementations, they result to similar performance in all the metrics measured. The modified version of `libpcap` that uses a separate thread for storing packets to the buffer seems to perform slightly better than the simple implementation.

We replayed the trace in different rates, from 10 to 300 Mbit/sec, trying different buffer sizes as before for each rate and we concluded to the same findings. In all rates, 4000 packets was found as the optimum buffer size. Using this optimum buffer size, locality buffering results in all rates to a significant reduction on Snort's cache misses and user time, similar to the improvement observed in 100 Mbit/sec against the original `libpcap`. The two implementations have almost equal performance in all cases, with the one using a thread performing a little better.

Another important metric for evaluating the improvement of our technique is the percentage of the packets that are being dropped in high rates by the kernel because Snort is not able to process all of the them in these rates. In Figure 6 we plot the average percentage of packets that are being

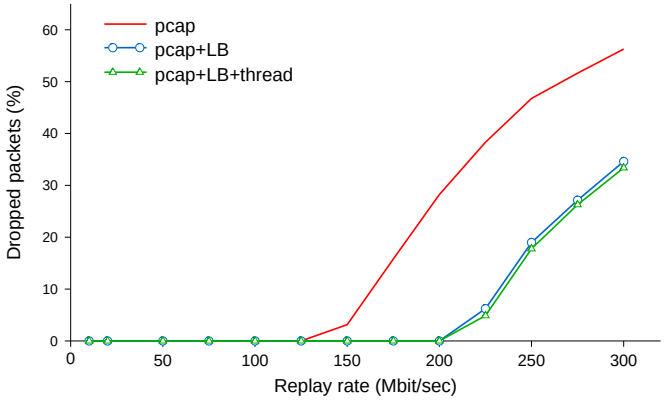

Fig. 6. Packet loss ratio of the passive monitoring sensor when running Snort, as a function of the traffic speed.

dropped while replaying the trace with speeds ranging from 10 to 300 Mbit/sec. We used 4000 packets size for the locality buffer, which was found to be the optimal size for Snort when replaying this traffic at any rate.

Using the unmodified `libpcap`, Snort cannot process all packets in rates higher than 125 Mbit/sec, 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, when deploying our locality buffering implementations in Snort, it becomes much more resistant in packet loss. It begins to loose packets at 200 Mbit/sec instead of 125 Mbit/sec, which is a 60% improvement. Also, at 250 Mbit/sec, our implementation drops 2.6 times less packets than the original `libpcap`. The two different implementations of the locality buffering technique achieve almost the same performance, with the thread-based implementation having slightly less dropped packets.

We do not observe any significant improvement with the thread-based implementation, compared to the simple locality buffering implementation, because the major benefit of our technique is the acceleration of packet processing due to improving memory access locality. Moreover, in the constant and high traffic rates that we generated in our experiments,

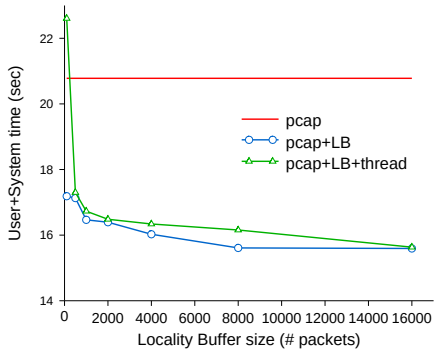

Fig. 7. Appmon's user plus system time as a function of the buffer size for 100 Mbit/s traffic.

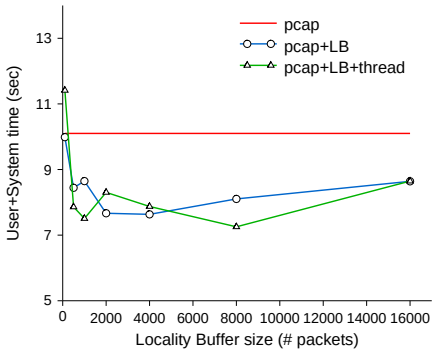

Fig. 8. Fprobe's user plus system time as a function of the buffer size for 100 Mbit/s traffic.

the CPU time was not idle during the packet gathering phase, since packets were continuously arriving. In case of bursty traffic, however, the separate thread would be more resistant to dropping packets.

B. 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 versions of `libpcap` and examined the improvement that they can offer using different buffer sizes that vary from 100 to 16000 packets. Figure 7 presents the Appmon's user plus system time measured while replaying the trace at a constant rate of 100 Mbit/sec.

The results show that the Appmon's performance can be improved using the locality buffering implementations. Its cache misses are reduced from 8.4 to 7.1 misses per packet, when used buffer size of 8000 packets, that is a 18% improvement. Thus, the user plus system time is reduced by more than 30% compared width the original `libpcap`. The optimum buffer size in the case of Appmon, based on the these results, seems to be around 8000 packets. Our different implementations resulted again to very close performance, with the first one giving a little better results this time.

We were also running Appmon when replaying traffic in rates varying from 10 to 300 Mbit/sec, observing always similar results. Since Appmon does significantly less processing than snort, no packets were dropped in these 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.

C. Fprobe

`Fprobe` [3] is a passive monitoring application that collects traffic statistics for each active flow and exports corresponding NetFlow records. We ran Fprobe with our modified versions of `libpcap` and performed the same measurements as with Appmon. Figure 8 plots the user plus system time of the Fprobe variants per buffer sizes from 100 up to 16000 packets, while replaying the trace at 100 Mbit/sec rate.

We notice a speedup of about 30% when locality buffering is enabled. The buffer size that optimizes overall performance is again around 8000 packets. We notice that in Appmon and Fprobe tools the optimum buffer size is about 8000 packets, while in Snort 4000 packets size is enough to optimize the performance. This happens because Appmon and Fprobe are not so CPU-intensive as Snort, so they require a larger amount of packets to be sorted in order to achieve a significant performance improvement. Finally, we observe that the version of `libpcap` that uses a separate thread for storing packets gives better performance in Fprobe for some of the buffer sizes, but it is not clear which of these two versions is preferable in this case. Similar results were observed in all rates of the replayed traffic.

V. 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. [7], 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. [8] present techniques for improving request locality on a

Web cache, which results to significant improvements in the file system performance.

Finally, several research efforts [9], [10], [11] have focused on improving the performance of packet capturing through kernel and library modifications which reduce the number of memory copies required for delivering a packet to the application. In contrast, our 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.

VI. Conclusion

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

We described in detail the design and the implementation of locality buffering within `libpcap`, and presented our experimental evaluation using three representative CPU-intensive passive monitoring applications. The evaluation results showed that all applications gain a significant performance improvement, while the system can keep up with higher traffic speeds without dropping packets. Specifically, locality buffering resulted to a 40% increase in the processing throughput of the Snort IDS, while the packet loss rate was decreased by 60%. Using the original `libpcap` implementation, the Snort sensor begins loosing packets when the monitored traffic speed reaches 125 Mbit/sec, while using locality buffering, packet loss is exhibited when exceeding 200 Mbit/sec. Fprobe, a NetFlow export probe, and Appmon, an accurate traffic classification application, also exhibited a significant throughput improvement, up to 30%, even though 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 needing to modify existing applications.

Acknowledgments

This work was supported in part by the IST project LOBSTER funded by the European Union under contract number 004336, and in part by the project CyberScope funded by the Greek General Secretariat for Research and Technology under contract number PENED 03ED440. A. Papadogiannakis, D. Antoniades, M. Polychronakis and Evangelos P. Markatos are also with the University of Crete.

References

[1] M. Roesch, “Snort: Lightweight intrusion detection for networks,” in *Proceedings of the 1999 USENIX LISA Systems Administration Conference*, November 1999. [Online]. Available: http://www.snort.org

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

[3] “fprobe: Netflow probes.” [Online]. Available: http://fprobe.sourceforge.net/

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

[5] “Performance application programming interface.” [Online]. Available: http://icl.cs.utk.edu/papi/

[6] “Tcpreplay.” [Online]. Available: http://tcpreplay.synfin.net/trac/

[7] K. Xinidis, I. Charitakis, S. Antonatos, K. G. Anagnostakis, and E. P. Markatos, “An active splitter architecture for intrusion detection and prevention,” *IEEE Transactions on Dependable and Secure Computing*, vol. 03, no. 1, pp. 31–44, 2006.

[8] E. P. Markatos, D. N. Pnevmatikatos, M. D. Flouris, and M. G. H. Katevenis, “Web-conscious storage management for web proxies,” *IEEE/ACM Trans. Netw.*, vol. 10, no. 6, pp. 735–748, 2002.

[9] 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.

[10] P. Wood, “libpcap-mmap,” Los Alamos National Labs. [Online]. Available: http://public.lanl.gov/cpw/

[11] L. Deri, “Improving passive packet capture:beyond device polling,” in *Proceedings of SANE*, 2004.