A Network-Processor-Based Traffic Splitter for Intrusion Detection

Ioannis Charitakis, Kostas Anagnostakis, and Evangelos P. Markatos

ICS-FORTH Technical Report 342

September 2004

Abstract

Scaling network intrusion detection to high-speed networks can be achieved using multiple intrusion detection sensors operating in parallel coupled with a suitable load balancing traffic splitter. This paper examines a splitter architecture that incorporates two methods for improving system performance: the first is the use of *early filtering* where a portion of the packets is processed on the splitter instead of the sensors. The second is the use of *locality buffering*, where the splitter reorders packets in a way that improves memory access locality on the sensors. We have implemented our approach on top of an IXP1200 network processor and evaluated its performance using a combination of experimental evaluation and simulation. Our experiments suggest that early filtering reduces the number of packets to be processed by 32%, giving a 8% increase in sensor performance, while locality buffers improve sensor performance by 10%-18%.

Index Terms

network-level security and protection, Network Processors, Intrusion Detection

I. Charitakis and E. P. Markatos are with the Institute of Computer Science - Foundation for Research and Technology - Hellas. K.G. Anagnostakis is with the University of Pennsylvania. I. Charitakis and E. P. Markatos are also with the University of Crete. The work of K. G. Anagnostakis was done while at ICS-FORTH.

Parts of this paper's sections III and IV have appeared in the Proceedings of the IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Orlando Florida, October 2003.

I. Introduction

Network Intrusion Detection is receiving considerable attention as a mechanism for shielding our cyberinfrastructure against “attempts to compromise the confidentiality, integrity, availability, or to bypass the security mechanisms of a computer network” [3]. The typical function of a Network Intrusion Detection System (nIDS) is based on a set of *signatures* that describe known security threats: each signature corresponds one threat. During its operation, a nIDS examines all network packets against all threat signatures and determines whether any signatures indicating intrusion attempts have been matched.

Effective Intrusion Detection systems require significant amounts of computational resources. Indeed, widely deployed open-source nIDS such as *snort* [14] often need to match packet headers against tens of rules and packet payloads against many hundreds of strings defining attack signatures. This task of header and payload matching is much more expensive than the typical header processing performed by traditional network elements such as packet forwarders and firewalls. Indeed, recent research results suggest that state-of-the-art processors running the *snort* nIDS are able to monitor network links whose capacity does not exceed 100-150 Mbps [1], [6]. Therefore, performing Intrusion Detection at high network speeds, such as 1 Gbit/s and beyond, requires the employment of a more scalable solution, such as the use of multiple Intrusion Detection sensors operating in parallel, fed by a suitable traffic splitter.

The characteristics of Intrusion Detection place certain constraints on the design of such a traffic splitter. For instance, as observed by Kruegel *et al.* [9], packets that are part of a given attack context should be processed by the same Intrusion Detection sensor. Otherwise, the attack packets will be distributed over several different sensors, making it difficult, and computationally expensive, to combine information from the different sensors and recognize the attack. On the other hand, placing such restrictions on the mapping of packets to particular sensors may easily lead to load imbalance among sensors, resulting in overloaded sensors which can not cope with their workload. Actually, this load imbalance may be exploited by attackers in order to evade detection by means of overloading individual sensors and slipping the attack through them.

Given the high, and sometimes conflicting, resource demands of Intrusion Detection, we consider ways of boosting sensor performance by rethinking the design of nIDS traffic splitters. We argue that traffic splitters should implement more active operations on the traffic stream with the goal of reducing the load on the sensors, rather than just passively providing generic, flow-preserving load distribution.

This paper presents two such active mechanisms. The first is based on the observation that a significant fraction of packets only require header processing. Given that header processing is relatively cheap (and can be easily performed in hardware) we can implement this function as part of the splitter. The main benefit of this method of *early filtering* is that the amount of traffic that needs to be transmitted and processed by the sensors can be reduced significantly. The second mechanism is based on the observation that different types of packets trigger different subsets of the nIDS ruleset, placing a significant burden on the sensor memory architecture (*i.e.* reducing memory access locality). We present an algorithm for *locality buffering*, so that packets of the same type are grouped together on the splitter before being forwarded to the sensors. The benefit of this method is that it increases performance without altering the semantics of the traffic stream and without requiring changes on the sensors. We argue that the algorithm requires a reasonable amount of additional buffer memory and a small number of operations on each packet and can thus be be efficiently implemented as part of the splitter.

We evaluate the performance of our approach using a combination of simulation and experimental evaluation on top of a IXP1200 network processor. Our results suggest that early filtering reduces the number of packets to be processed by the end sensors by 32% giving an 8% increase in sensor performance, while locality buffering improves sensor performance by 10-18%.

A. Paper organization

The rest of this paper is organized as follows. In Section II we provide a brief overview of how a nIDS works and how load balancing is used for building scalable Internet services, including Intrusion Detection. In Section III we present a nIDS load balancing architecture implementing the proposed early filtering and locality buffering policies. In Section IV we present experiments examining the performance of the proposed methods. In Section V we discuss the implementation of our approach on top of the IXP1200 network processor, and in section VI evaluate its performance. Finally, in Section VII we summarize and conclude the paper.

II. Background

A. Network Intrusion Detection

In this section we describe a (simplified) model of how a Network Intrusion Detection System (nIDS) operates. A nIDS examines all network traffic against a set of signatures which describe the known attacks, and determines whether any packets match any signatures, which is a sign of possible Intrusion attempt. The simplest and most common form of nIDS inspection is to perform protocol header analysis and match string patterns against the payload of packets captured on a network link. Known systems following this model are *snort* [14] and *Bro* [13]. The following is a simplified *snort* signature for the CODE-RED worm which was released on the Internet during the summer of 2001:

```
OUT_NET any -> WEB_SERVER 80 content "GET /default.ida?NNNNN"
```

The above signature suggests that any packet coming from the network outside the organization (`OUT_NET`) on `any` source port, and is destined to the `WEB_SERVER` on port `80`, and contains the substring “`GET /default.ida?NNNNN`”, then it probably contains the CODE-RED attack and should be logged.

A nIDS is usually built as a passive monitoring system that reads packets from a network interface through packet capture facilities such as `libpcap` [11]. After being delivered by `libpcap`, each packet is checked against the nIDS *ruleset*. A ruleset is typically organized as a two-dimensional chain data-structure, where each element - often called a *chain header* - tests the input packet against a packet header rule. When a packet header rule is matched, the chain header usually points to a set of signature tests, including payload signatures that trigger the execution of a string matching algorithm [1], [5], [6].

B. Load balancing

Splitting network traffic among different servers, otherwise known as *traffic load balancing*, has been widely for building scalable Internet services, and has been applied to widely deployed systems such as web servers [4], [7].

Recently, researchers have started to examine a general approach for load balancing as applied to high speed Intrusion Detection systems. For example, Kruegel et al. [9] propose an two-stage architecture for determining the set of sensors that will process each packet of the network traffic. The first stage of the architecture attempts to equally distribute packets, while the second stage examines packets for determining a suitable set of sensors for final processing. The decision of where to send a packet is based on rules describing the attack contexts to which a packet may belong. The main focus of that work is therefore to preserve detection semantics in a generalized model of Intrusion Detection, assuming different types of detection such as statistical methods, anomaly detection and content-matching. In contrast, our work is performance-oriented and focuses on the specific case of content-matching Intrusion Detection as widely deployed today. In addition to research prototypes, commercial nIDS load balancing products have recently started to become available, such as [15]. Unfortunately, such products provide little, in any, information publicly available about the details of their design as well and the load balancing policies they employ.

C. Early Filtering and load balancing

The idea of providing filtering functionality on a load balancer is also discussed in [7], where the splitting device is instructed to block traffic destined to unpublished ports. Although the functionality proposed by Goldsmidt and Hunt [7] is similar to the functionality provided in our work, the goals are different: our goal is to enhance sensor performance, not provide firewall-like protection from irrelevant or malicious traffic.

D. Locality enhancing techniques

Locality enhancing techniques for improving server performance are well studied. For example, in [10] the authors try to improve request locality on a Web cache, demonstrating significant improvements in file system performance. However, to the best of our knowledge, this current paper is the first attempt to providing locality enhancements as part of a load balancer, and the first to do so in the context of Intrusion Detection.

III. DESIGN

There are four main goals in designing a nIDS traffic splitter. First, packets that belong to the same attack context need to be processed by the same Intrusion Detection sensor. Otherwise certain attacks would not be detected. For content-based intrusion detection this can be achieved

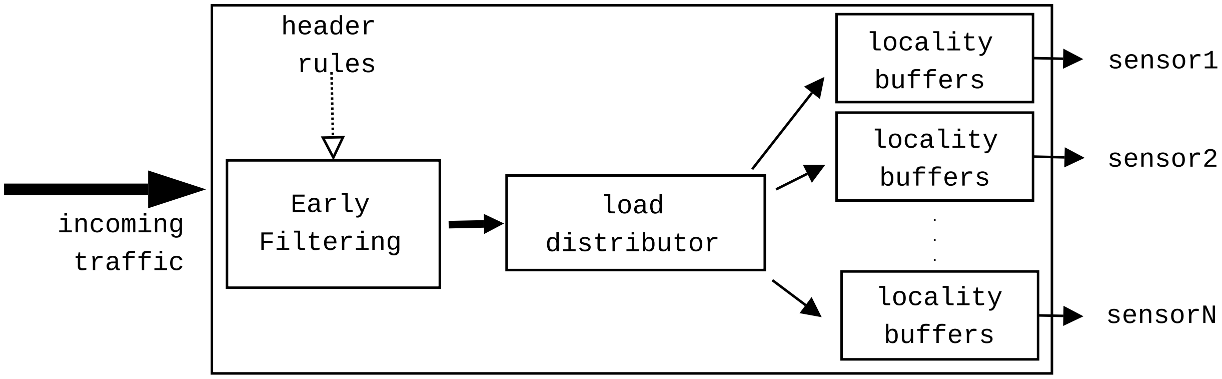

Fig. 1. The active nIDS splitter architecture

by mapping packets of the same flow to the same sensor. Second, traffic should be distributed so that overall system performance is maximized. Assuming a set of N identical sensors (in terms of resources, software and configuration), a good way of achieving this is to distribute approximately $1/N$ of the total load to each sensor. Flow-level traffic distribution works well toward this goal, and we will discuss how early filtering and locality buffering can provide further benefits. Third, the load balancing algorithm needs to be efficient enough to operate at high network speeds. Fourth, the system should (ideally) not require modification to the sensor function.

The overall architecture architecture of our approach is shown in Figure 1. All incoming network traffic arrives from the left side of Figure 1 and enters the traffic splitter. The splitter, after some early-filtering pre-processing, divides the traffic through the load balancer into separate streams and sends each of them to a different sensor which processes the incoming packets searching for possible intrusion attempts. Given that the end sensors are off-the-self Intrusion Detection Systems, such as snort [14] our contribution is focused on the architecture and implementation of the traffic splitter.

As shown in Figure 1 the system is composed of an early filtering element, a load distribution element and a set of locality buffering units, one unit for each sensor. In the remainder of this Section, we will present each of the elements in more detail.

A. Early filtering

The goal of early filtering is to identify those incoming packets that do not contain any intrusions and filter them out immediately without needing to sending them to the end sensors. Such an early filter will reduce the load on the end sensors, and may also improve the performance of the overall system, as the process of sending the filtered-out packets from the splitter to the sensors will be avoided.

To perform early filtering, we analyzed the *snort* ruleset and found 165 rules that require only header (not payload) processing: we refer to this set of rules as the *EF ruleset*.

Once the *EF ruleset* has been identified, the splitter then operates as follows: when a packet is received, it is first checked against the EF ruleset. If (i) no rule is matched and (ii) the packet contains no payload, then the packet is filtered out (i.e. discarded). Otherwise, it is forwarded to the end sensors for further processing. Note that the packets which are forwarded to the end sensors may belong to one of the following two classes: (i) they matched one of the rules from the *EF ruleset*, or (ii) they did not match any of the *EF ruleset* rules, but they contain payload. Packets belonging to the first class are forwarded to the end sensors in order to be logged, while packets belonging to the second class are forwarded to the end sensors in order to be examined against the rest of the *snort*'s rules.

B. Load distribution

The goal of load distribution is to divide the network traffic among the end sensors so as to keep them as evenly loaded as possible. At the same time, the distribution of the network traffic should make sure that all packets of a network flow are examined by the same sensor, otherwise the system may miss an intrusion attempt.

A simple and efficient approach for load distribution is to compute a hash function on some of the fields of the packet headers, and to assign each packet to an end sensor based on the resulting value if this hash function. A hash function such as CRC16 [4], can evenly spread the flows among the sensors, so that each sensor will receive an approximately equal amount of work. Careful choice of the header fields that will be used as input to the hash function can result in a load balancing policy that is flow preserving, i.e. packets of the same flow will be assigned to the same sensor. This can be easily accomplished by using the following header fields: protocol number, source IP address, destination IP address, source port, and destination

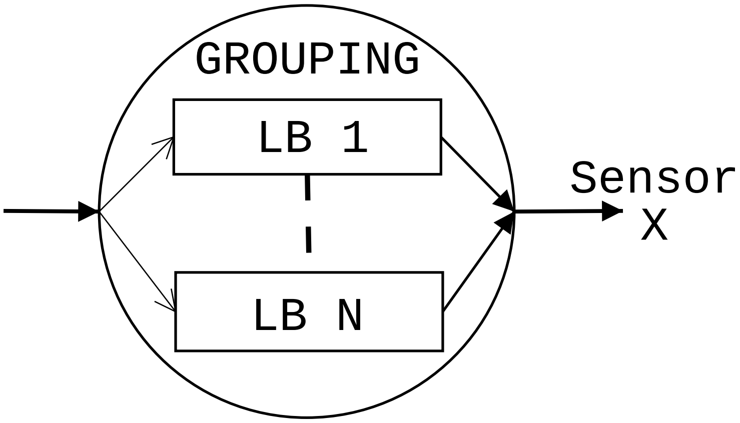

Fig. 2. Packet grouping using Locality Buffers

port. Assuming well-behaved (*e.g.* TCP-friendly) traffic, this approach is also robust to variations in traffic load, as new flows will be assigned evenly among the available sensors. Of course, such an approach may not be robust against attackers attempting to overload the system to evade detection. However, this problem is beyond the scope of this work and is also not specific to the proposed enhancements.

For the purpose of our study we have used a CRC16-like hashing function as it has been shown to perform well [4].

C. Locality buffering

Locality buffering is a method for adapting the packet stream in such a way that improves performance of each end Intrusion Detection sensor by reducing its cache misses and improving the locality of its memory accesses.

The observation on which locality buffers are based is the following: each packet that arrives at the end sensor will be checked against rules that apply to the packet's type of traffic. For example, network packets destined to a web server will be checked against a set of rules which search for web server exploits. Similarly, network packets destined to an ftp server will be checked against a set of rules which describe ftp server exploits. When checking a packet against a set of rules, each sensor will have to bring this ruleset to the first-, and possibly the second-, level cache of the processor on top of which it executes. In an ordinary traffic stream, packets from different network flows appear interleaved. For example, assume an Intrusion Detection sensor which monitors a traffic stream consisting of packets belonging to a web flow and packets belonging to an ftp flow. Ordinarily, web packets will arrive interleaved with ftp packets, which implies that the sensor will alternate in its cache the web ruleset with the ftp ruleset, resulting in cache misses and reduced performance.

To increase memory locality and reduce cache misses, locality buffering attempts to rearrange the interleaving of packets in the network traffic so that packets that arrive back-to-back will trigger the same ruleset as frequently as possible. To do so, our method uses a set of *locality buffers*. Instead of sending network packets to end Intrusion Detection sensors directly, our approach places packets in locality buffers, so that packets placed in the same buffer will trigger the same ruleset in the Intrusion Detection sensor. When a buffer becomes full, all its packets are transmitted back-to-back to the target sensor. Therefore, packets arriving back-to-back at an end sensor will have a higher probability of triggering the same ruleset and of improving the memory locality.

Since exact classification of each packet according to the nIDS rule-groups can be complicated, we have opted for a simpler solution based on the following heuristics for determining the target locality buffer for a given packet:

SD: We place a packet in a locality buffer based on the result of a hash function computed on the source and the destination ports of the packet. Using this approach we expect that packets belonging to different flows will end up in different buffers, thereby reducing packet interleaving.

D: We place a packet in a locality buffer based on the result of a hash function computed on the destination port only.

T_D: In this approach we allocate a subset of locality buffers for known traffic types and use method D for the remaining buffers/packets. For example, one buffer may receive only Web traffic, another buffer may receive only NNTP traffic, and a third buffer may receive only P2P traffic. Unclassified packets are then allocated to the rest of the locality buffers using method D: hashing on the destination port only. The choice of traffic types can be made by profiling real network traffic and looking at how the nIDS ruleset is utilized.

TABLE I

LOCALITY BUFFER ALLOCATION METHODS

Method	Description
SD	hash(*S*rc+*D*st port)
D	hash(*D*st port)
T_D	Dedicate LBs to specific traffic *T*ype + method *D*

IV. PERFORMANCE OF EARLY FILTERING AND LOCALITY BUFFERING

In this section we present experiments examining the effect of early filtering and locality buffering on NIDs performance.

For our experiments we use a Dell PowerEdge 500SC equipped with a 1.13 GHz Pentium III processor PC with 8 KB L1 cache, 512 KB L2 cache and 512 MB of main memory. The host operating system is Linux (kernel version 2.4.17, RedHat 7.2). The NIDs software is *snort* version 2.0-beta20 compiled with `gcc` version 2.96 (optimization flags `O2`).

All experiments are performed by reading a packet trace from disk, except for the early filtering experiments where traffic is received from the network (to capture the effect of early filtering on the network subsystem). In the later case, we use a simple network with two hosts A and B and a monitoring host S. Host A reads the trace from file and sends traffic to host B (using `tcpreplay`) over a 100 Mbit/s Ethernet switch configured to mirror the traffic to host S. As the exact timing of trace packets has negligible effect on NIDs behavior, we simply replay the trace at maximum rate (link utilization was roughly 90%).

We drive our experiments using the `nlanr.MRA.1031627450` packet trace from the NLANR archive captured in September 2002 on the OC12c (622 Mbit/s) PoS link connecting the Merit premises in East Lansing to Internet2/Abilene [12]. The trace contains 2,760,531 packets with average size of 762 bytes. Of these packets 96% are TCP packets, and 3.55% are UDP packets. Since the trace contains only the header (and not the payload) portion of each packet we added uniformly random payload data to create realistic traffic. [1]

[1] The use of random payloads for NIDs evaluation is shown in [2] to offer reasonably accurate performance estimates.

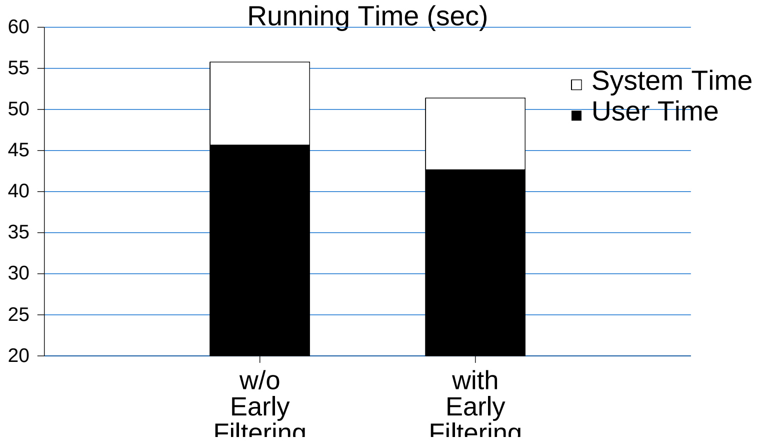

Fig. 3. The effect of Early Filtering on sensor performance.

A. Evaluation of Early Filtering

In our first set of experiments we set out to explore the benefits of early filtering. Analyzing the trace we used in our experiments reveals that more than 40% of the packets do not contain any payload - most of these packets are TCP acknowledgments. Moreover, more than 99% of these do not match any of the rules in the EF ruleset, and therefore, they can safely be dropped by the splitter during early filtering.

To measure the effect of early filtering on sensor performance, we measure the user and system time of running *snort* on top of two traces: First on top of the original trace, and second on top of a stripped trace which does not contain the packets that are dropped by early filtering. The results are depicted in Figure 3: the left bar of the figure shows snort's running time on top of the original trace, and the right bar is the running time on top of the stripped trace (i.e. early filtering). We observe that user time is reduced by 6.6% (45.67 sec vs. 42.66 sec) while system time is decreased by 16.8% (10.1 sec vs. 8.7 sec). Considering both user and system time the results suggest an overall improvement of 8%.

B. Performance of Load Sharing Hash Function

In this section we explore the load balancing properties of the CRC16 hash function which we use to distribute packets among the available sensors. For this purpose, we use the hash function to spread the packets among the available sensors and we measure the *maximum* number of

TABLE II

PERFORMANCE OF CRC16-BASED LOAD SHARING METHOD.

Sensors	difference (in % of assigned packets) of most loaded from fair share
2	1.25%
4	5.70%
8	13.55%

packets received by any sensor, as well as the *average* number of packets received by the sensors for the cases of 2, 4 and 8 sensors. Table II shows the percentage difference between the maximum and the average number of packets received by 2, 4, and 8 sensors. We see that this difference is rather small (i.e. 1.25%) for the case of two sensors, but it is noticeable (i.e. 13.55%), although not significant, for the case of eight sensors.

C. Effect of Locality Buffers on nIDs performance

To investigate the benefit of using locality buffers we measure the nIDS performance using two metrics:

- aggregate user time[2]: the total user time spent by all *snort* sensors.
- maximum user time: the user time spent by the most loaded sensor.

We determine how performance is affected when using different numbers of participating sensors, number and size of locality buffers, as well as different heuristics for locality buffer allocation.

1) Effect of Locality Buffers vs. Number of Sensors: Figure 4 shows the aggregate user time for different numbers of sensors, and Figure 5 shows the user time of the slowest (most loaded) sensor. For this set of experiments we use 16 locality buffers of 256 KB each and the T_D allocation method. Figure 4 shows that using locality buffers improves the aggregate user time by at least 11.4% (8 sensors) and up to 13.8% (one sensor). Figure 5 shows that using locality

[2] We have excluded the system time from our metrics, since it consists only of kernel overhead related to reading the network packets from the trace stored on disk.

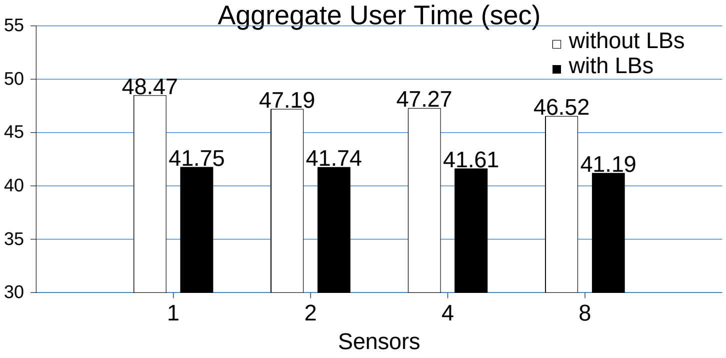

Fig. 4. Aggregate user time over all sensors vs. number of sensors.

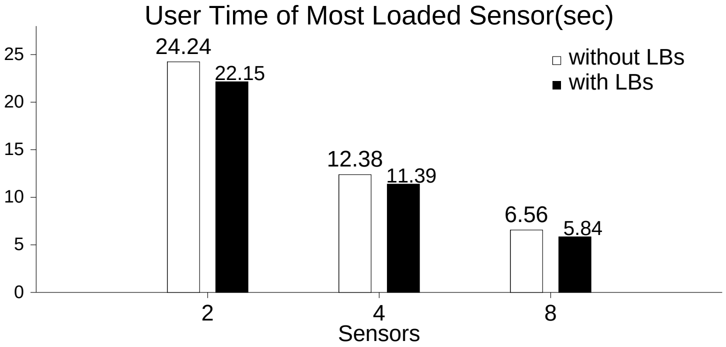

Fig. 5. User time of slowest sensor vs. number of sensors for the experiments of Figure 4

buffers improves the time of the most loaded sensor by 9%-12%. One interesting observation from Figure 4 is that as the number of sensors increases, the aggregate user time (in white bars) is decreasing. This happens because distributing packets to a large number of different sensors, even in the absence of locality buffers, demultiplexes the incoming traffic and increases the probability of same-type back-to-back packets. To verify this observation we measure the average burst size (*i.e.*, the number of consecutive packets that have the same protocol and the same destination port) seen by the sensors in the experiments of Figures 4 and 5. Figure 6 presents the average burst size for one to eight sensors. By looking at Figure 6, it is evident that the average burst size increases with the number of sensors. For example, in the absence of locality buffers the average burst size increases from 1.06 packets to 1.18 packets, an 11% increase. Similarly, when locality buffers are being used, the average burst size increases from

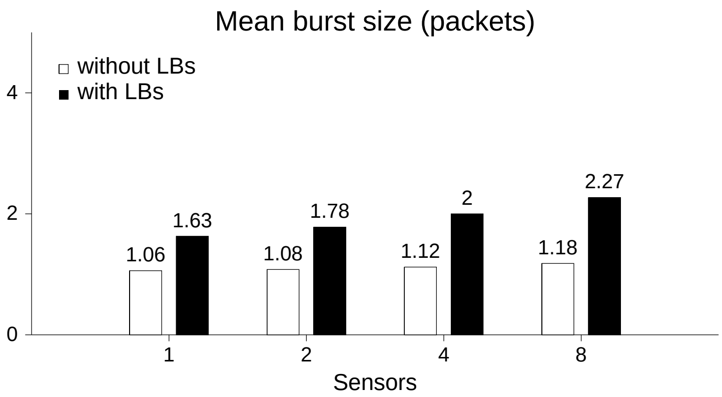

Fig. 6. Mean burst size vs number of sensors for the experiment of Figures 4 and 5.

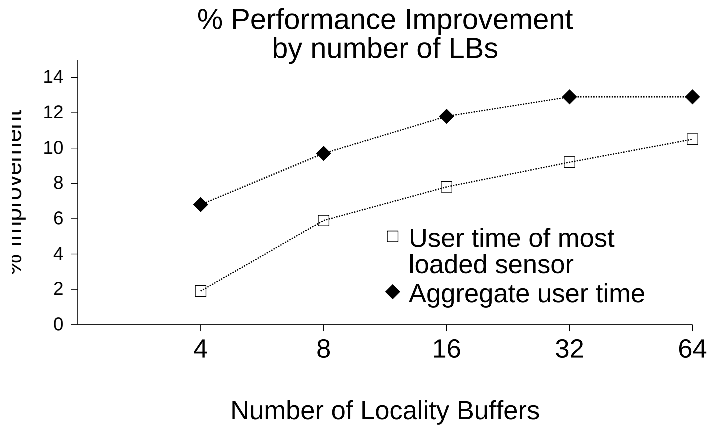

Fig. 7. Performance improvement (reduction in user time) using different number of LBs.

1.63 to 2.27, a 39% increase. It is interesting, however, to note that the average burst size in almost all cases increases significantly with the use of locality buffers. For example, in the case of one sensor locality buffers increase the burst size by 53% (from 1.06 to 1.63), and in the case of eight sensors by 92% (from 1.18 to 2.27).

2) Locality Buffer dimensioning: In our next set of experiments we investigate how the size and the number of locality buffers affect performance. We use four sensors and the locality buffers are allocated using method T_D. In each experiment we measure the difference in user time compared to a system without locality buffers.

Figure 7 shows the results of using different number of locality buffers per sensor when the

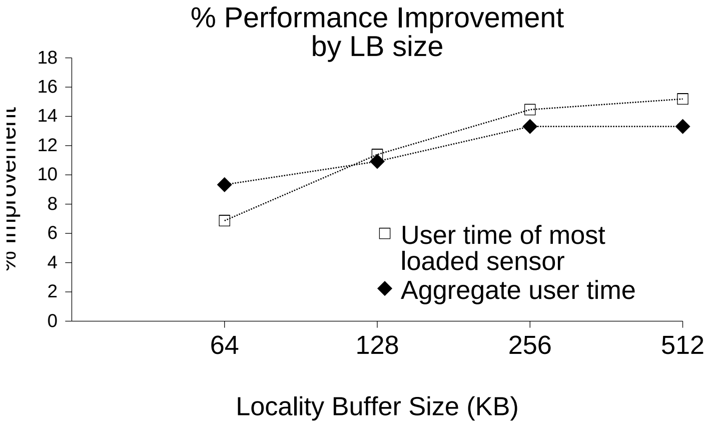

Fig. 8. Performance improvement (reduction in user time) using different size for each LB.

size of each buffer is 256 KB. We observe that the improvement in aggregate user time varies between 6.8% (4 buffers) and 12.9% (64 buffers). Increasing the number of locality buffers beyond 32 does not appear to offer any benefit in terms of aggregate user time although it still improves the performance of the most loaded sensor. This suggests that using 32 or 64 locality buffers per sensor is a reasonable design choice.

To measure how the size of each locality buffer affects performance we measure the aggregate user time and the user time of the most loaded sensor for various buffer sizes. The results are presented in Figure 8. The reduction in aggregate user time ranges from 9.3% to 13.31% for the cases of 64 KB and 512 KB respectively. Using 256 KB per locality buffer seems like a reasonable choice, as the gain of increasing the buffer size from 256 KB to 512 KB is marginal.

3) Effect of different locality buffering policies: In this next set of experiments we examine how the different heuristics for allocating locality buffers affect performance. For this set of experiments we use four sensors, 16 locality buffers per sensor and 256 KB per buffer. Again, we measure the percentage of reduction locality buffers achieve in aggregate user time.

Figure 9 shows the performance improvement in terms of aggregate user time as well as user time of the slowest sensor, for different locality buffer allocation methods. We see that using hashing on the destination port only (*D*) is better than simple hashing on both ports (*SD*) by more

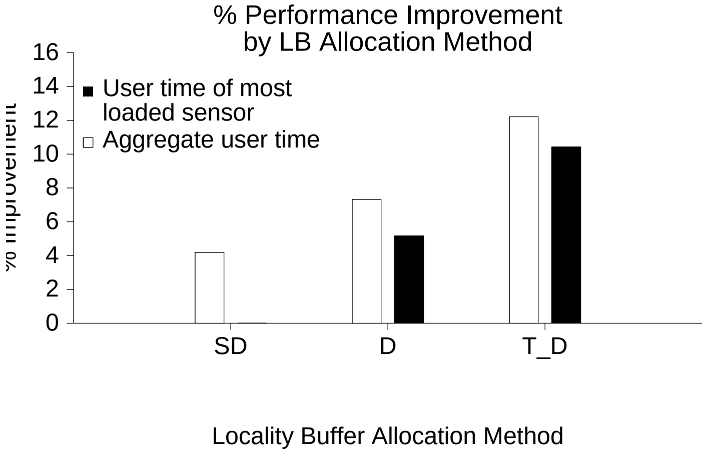

Fig. 9. Percentage of performance improvement when using different Locality Buffer Allocation Methods.

TABLE III

EVALUATION OF EARLY FILTERING AND LOCALITY BUFFERING.

	Aggregate		Most Loaded Sensor	
	time (sec)	improvement	time (sec)	improvement
base system	47.27		11.52	
locality buffers	41.61	8.9%	10.93	5%
locality buffers + early filtering	37.88	19.8%	10.06	14.4%

than 4%. We also see that the best performance is obtained when assigning some of the locality buffers to specific types traffic. This is observed in bars *T_D* which shows an improvement of 12.19%. This is not surprising, as a significant part of the trace includes Web traffic, and therefore dedicating buffers to this kind of traffic results in longer bursts of similar packets.

D. Evaluation of Early Filtering combined with Locality Buffers

To estimate the benefits of using both early filtering and locality buffering together we apply the early filtering method on the packet trace and split the remaining packets to four sensors using 16 Locality Buffers of 256 KB per sensor and buffer allocation heuristic *T_D*. Table III shows the results. The measured aggregate user time is 37.88 sec compared to 41.61 sec when using locality buffers only, reflecting an improvement of 8.9%. Compared to 47.27 sec when not using locality buffers at all, the overall improvement of using both EF and LB is 19.8%. For the slowest sensor, performance is increased by 5% when compared to using only locality buffers (from 11.52 sec to 10.93 sec) and 14.4% when compared to not using early filtering or locality buffers.

V. Implementation of Splitter on the IXP 1200

We have implemented our approach on top of the Intel IXP1200 network processor [8]. The IXP1200 network processor is equipped with one general-purpose StrongArm processor and six special-purpose processors which are usually called micro-engines (uEngines). Each uEngine is equipped with four threads which frequently context switch among themselves in order to mask memory latency. In our experimental environment, the IXP1200 network processor is mounted on an ENP-2506 development board provided by Radisys. Besides the processor, the board includes 256 MB of SDRAM, 8 MB of SRAM, 8 MB of Flash memory, two optical gigabit interfaces and a 64 bit external PCI interface. The IXP 1200 network processor is internally clocked at 232 MHz.

A. Splitter Overview

Figure 10 outlines the main components of our software running on the IXP1200 network processor. Each incoming packet entering the network processor (right side of Figure 10), is assigned to a target Intrusion Detection sensor which will check the packet for possible cyberattacks. Sensor assignment is performed in a flow preserving manner, i.e. all packets of the same flow will always be assigned to the same sensor. This is accomplished by assigning packets to sensors based on the result of a hash function applied on the source and destination IP addresses of the packet. Following sensor assignment, each packet is assigned to one of 16 locality buffers (dedicated to each sensor) based on the result of a hash function computed on the packet's destination port. An exception to this rule are packets belonging to the following traffic categories:

- packets destined to port 80 (web client traffic)
- packets originating from 80 (web server traffic)
- packets destined to port 119 (nntp traffic)
- packets destined to port 1214 (Kazaa traffic)

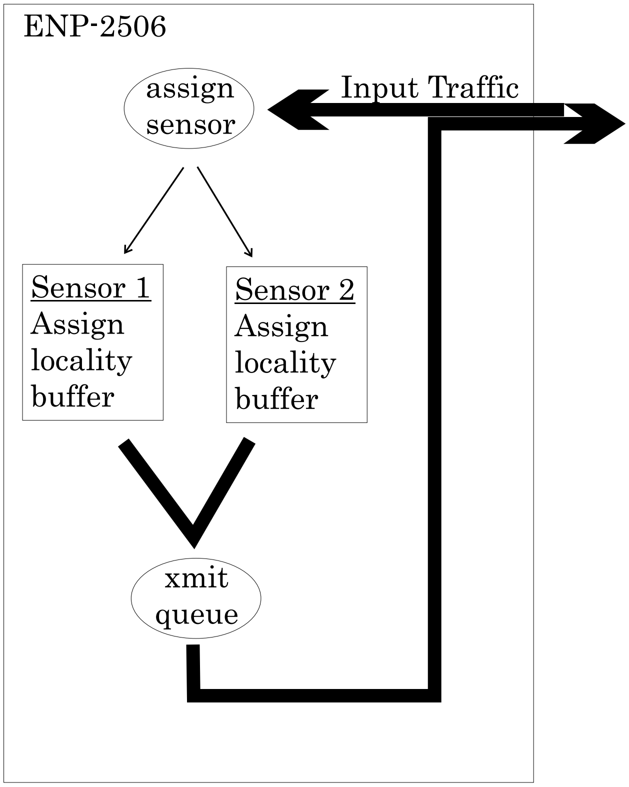
ENP-2506
assign sensor
Input Traffic
Sensor 1 Assign locality buffer
Sensor 2 Assign locality buffer
xmit queue

Fig. 10. Splitter work overview

- packets destined to port 445 (Common Internet File System)

Each of the above categories is given its own locality buffer.

When a locality buffer becomes full, all its packets are enqueued in the transmit queue (xmit queue at the bottom of Figure 10), and transferred to the final sensor as a single burst (back to back).

Figure 11 shows the specific work of each programmable element inside the IXP, as well as how the memory resources are used. As it can be seen the distribution of work is as follows:

- **Receive Threads** There are two uEngines, a total of eight Receive Threads, which receive packets, assign them to the corresponding sensor and maintain the locality buffers by enqueueing packets to and dequeuing packets from the buffers. Dequeued packets are subsequently enqueued in the xmit queue.
- **Transmit Threads** One uEngine is dedicated to transmission using three Transmission Threads and one Transmission Controller Thread.

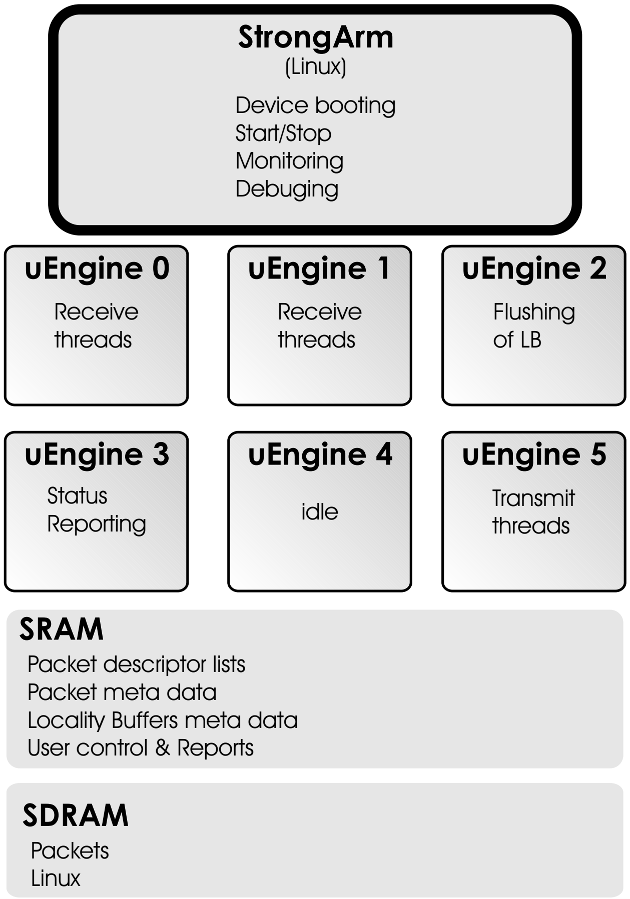

Fig. 11. Distribution of work and data among the components of IXP

- **Flushing of locality buffers** One uEngine is dedicated to periodically checking the contents of the locality buffers and flushing them when a timeout period has elapsed.
- **Status Reporting** One uEngine is dedicated to periodic gathering of statistics.
- **StrongArm** The StrongArm general-purpose processor is dedicated to running Linux, and it is used for booting and performing the primary initialization of the device. Moreover, it is used for starting and stopping the IXP Splitter operation, for monitoring its state, and for debugging purposes.

It can be easily seen that our design utilizes only five of the available six uEngines leaving head room for future expansions. The available memory of the IXP1200 is allocated as follows:

- **SDRAM** 32 MB of SDRAM are dedicated to storing the actual contents of each packet. Of the rest of the 256 MB of the SDRAM, about 64 MB are occupied by the Linux Operating

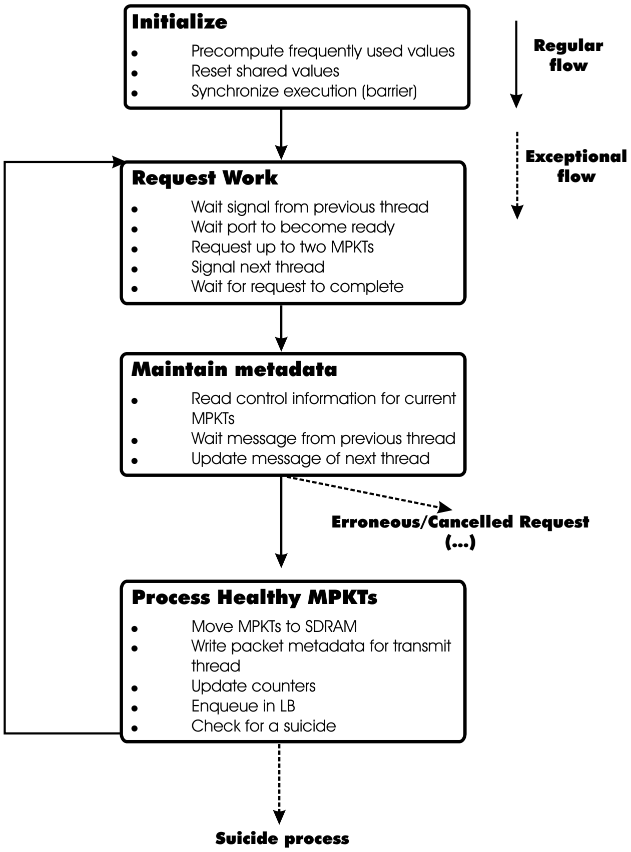
Initialize
Precompute frequently used values
Reset shared values
Synchronize execution (barrier)
Regular flow
Exceptional flow
Request Work
Wait signal from previous thread
Wait port to become ready
Request up to two MPKTs
Signal next thread
Wait for request to complete
Maintain metadata
Read control information for current MPKTs
Wait message from previous thread
Update message of next thread
Erroneous/Cancelled Request (...)
Process Healthy MPKTs
Move MPKTs to SDRAM
Write packet metadata for transmit thread
Update counters
Enqueue in LB
Check for a suicide
Suicide process

Fig. 12. Steps executed by each Receiver Thread

System, and the rest is mainly unused.

- **SRAM** 2 MB of SRAM are dedicated to :
  - Lists of packet descriptors.
  - Per packet meta data.
  - Per locality buffer meta data.
  - User control and reports.
  - Synchronization of code and system variables.

B. Receive Procedure

Figure 12 outlines the algorithm executed by each of the Receive Threads.

1) Initialization: During initialization we pre-compute and store in registers several commonly used values in order to reduce run-time overhead. Such values include the configuration of the

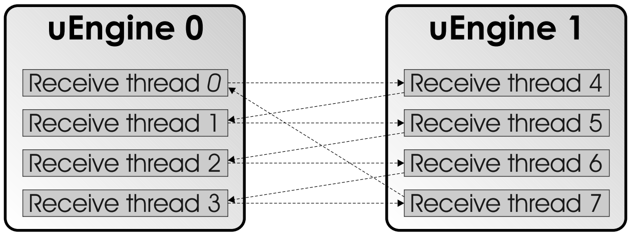

Fig. 13. Order of execution for the receive threads

receive request, the configuration of signalling the next thread, and the configuration of a spare packet descriptor.

2) Request Work: Figure 13 shows the order in which the Receive Threads issue requests for work (receive requests). The scheme is interleaved so as to balance the load from the start up.

After receiving the signal from the previous thread, each Receive Thread polls the "fast receive ready" flag of the monitored port. When enough data are available, it issues a receive request for two MPKTs [3] maximum number of bytes to be transferred to the receive buffer.

3) Metadata maintenance: Note that the value of "fast receive ready" flag is slightly outdated due to the small latency between polling the flag and actually getting the MPKT. Therefore when the receive request will be serviced by the Receive State Machine, it may end up:

- to a canceled receive request.
- to transfer one MPKT instead of the requested two.
- to Transfer two MPKTs.

Depending on whether the request was canceled or not, the thread will proceed either to Requesting Work or processing the Healthy MPKT.

4) Process MPKTs: In the later case, the Receive Thread will have to move the corresponding MPKT to the correct place in the SDRAM. If this MPKT was flagged as SOP (Start Of Packet), then the Receive Thread will use its spare descriptor in order to decide where in SDRAM will be placed. Otherwise, the Receive Thread will have to know where the previous MPKTs where stored so as to continue.

To accomplish that, Receive Threads communicate by writing messages to specific places in SRAM the *message boxes*. The structure of each message box is depicted in Figure 14. Each

[3] An MPKT is a 64-byte packet fragment.

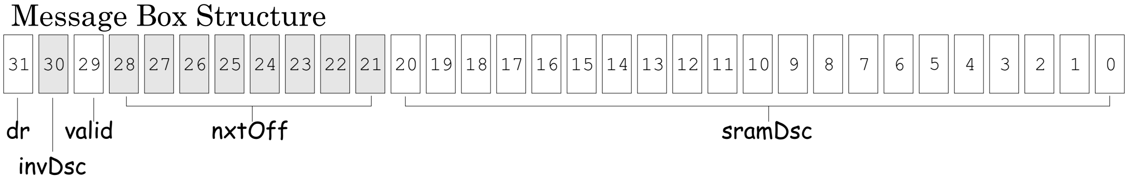

Fig. 14. Structure of *message box*

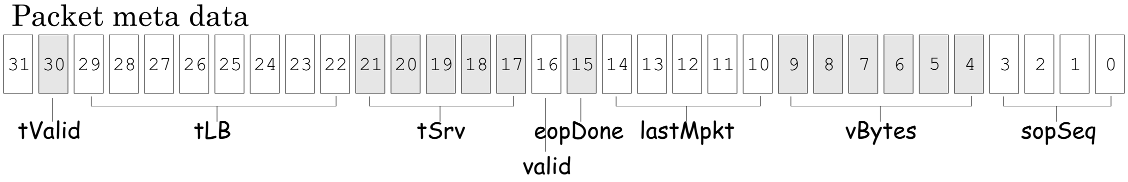

Fig. 15. Packet Meta Data

Receive Thread initially waits for its own message box to be validated by the previous thread. Then it reads and invalidates its contents, decides how to update them, and writes the updated message in the box of the next thread. Finally it validates that box. In this way, each Receive Thread knows where in SDRAM the current MPKT belongs and at what exact offset. Moreover, packets are serialized and order maintained.

After transferring the MPKT to SDRAM, counters are updated and in the case of End Of Packet, the packet meta data (Figure 15) are also updated. In the case of SOP, the packet is assigned using hashing to one of the target sensors. It is also assigned to corresponding locality buffer. These assignments are stored in the packet metadata so that the Receive Thread processing the EOP MPKT will get them ready.

Therefore in the case of EOP, the Receive Thread will read and update the packet meta data and will enqueue the packet to the correct locality buffer for the correct Sensor. The corresponding

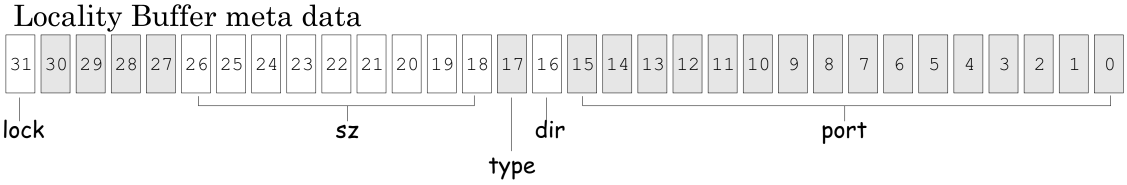

Fig. 16. Locality Buffer meta data structure

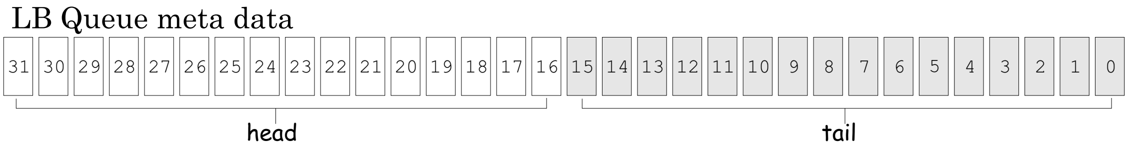

Fig. 17. Meta data of each queue holding the contents of one Locality Buffer

meta data will be updated (Figures 16 and 17). If the corresponding locality buffer becomes full, it will enqueue all the packets in the xmit queue for transmission. This requires only the update of the tail pointer of the xmit queue.

C. Transmit Procedure

1) Overview of Transmit Procedure: Three threads of the Transmit uEngine (the *Transmit Threads*) cooperate for the transmission of each packet. Each of these threads handles the transmission of one or two MPKTs belonging to the packet under transmission. These threads execute the same code which is outlined bellow:

1) Reserve the next one or two MPKTs for transmission.
2) Move the reserved MPKT(s) from SDRAM to Transmit Buffer.
3) Set the correct control information about the transmission of the specific MPKT(s).
4) Wait for the previous thread to finish the transmission of previous MPKTs.
5) Wait device to become ready.
6) Validate the Transmit Buffer (initiate transmission).
7) If there are no more MPKTs for this packet, push the corresponding descriptor back to the stack of free descriptors.

2) Shared Transmission Data Structure: The synchronization of the Transmit Threads is accomplished via a shared structure describing the current packet under transmission and its exact transmission state. This shared structure (*packet_xfer_data*) has the following fields:

- sdramAdd: Address in sdram from where the packet data start.
- **dsc:** Descriptor reserved for this packet.
- **firstMpktNo:** Number of the first MPKT of this packet. MPKTs are numbered consequently, and the numbering spans over the transmitted packets.
- **lastMpktNo:** Number of the last MPKT belonging to this packet.

- **pktMetaData:** Meta data of this packet. This is a copy of the structure of Figure 15.
- **lastMpktValidBytes:** Number of valid bytes in the last MPKT.

Transmit Threads lock, read, update and unlock this shared structure. All these operations are fast since all the threads are executing within the same uEngine, and more over thread execution is non-preemptive. However, updating this data structure with a new packet is rather slow since, among other operations, the packet has to be dequeued from the xmit queue. This is why a separate thread handles this operation. The function of the *Transmission Controller* is outlined in the following paragraph.

3) Transmission Controller Thread: The Transmission Controller Thread executes in the same uEngine as the rest of the Transmission Threads, making the synchronization with the Transmit Threads simpler. The main purpose of this thread is to hide the latency of dequeuing the next packet for transmission by overlapping with the operation of transmitting the current packet. Its function is described below:

1) Read the next packet to transmit
    a) Read meta data of xmit queue.
    b) Get packet pointed by the Head of xmit queue.
2) Prepare the *packet_xfer_data* for the recently dequeued packet.
3) Wait for current packet transmission to complete.
4) Set the prepared *packet_xfer_data* as current.
5) Repeat from the beginning.

Therefore, while the Transmit Threads are busy transmitting the current packet, the Transmit Controller prepares the next packet for transmission

D. Synchronization between Receive Threads and Transmit Controller

Communication of Receive Threads and Transmit Threads is accomplished via the shared xmit queue. Receive Threads enqueue packets for transmission in this queue and the Transmit Controller dequeues. New packets are placed on top (in front of the Head), while the Transmit Controller dequeues from the Tail. Consistency is maintained by restricting the transmit controller *not* to dequeue (update the Tail) when the Head and the Tail point to the same packet, or when the xmit queue is empty.

TABLE IV

TRANSMISSION CAPACITY OF THE IXP1200-BASED SPLITTER

packet length	throughput achieved (Mbps)
64	500
1472	980

VI. NETWORK PROCESSOR SPLITTER EVALUATION

In this section we report the evaluation of the network-processor-based implementation of the splitter and the locality buffers.

A. Performance of Packet Transmission

In this first experiment we measure the packet transmission capacity of the IXP-based splitter (table IV). To measure only the transmission capacity of the splitter, we disabled the packet receive functionality and as a result the splitter was transmitting the same packet over and over. For large packets (1472 bytes long), the splitter manages to achieve a transmit rate of around 980 Mbit/s, while for small packets (64 bytes long) the achieved rate was around 500 Mbit/s. These results suggest that using the Transmit Code alone, the IXP can be used as a simple packet generator for stress-testing the performance of other network elements.

B. Performance of Packet Reception

In this next set of experiments we evaluate the receiving capacity of the IXP1200-based splitter. In this setting we use one IXP1200 board as the traffic generator and another IXP1200 as the traffic sink. The traffic generator was generating 1472-bytes long packets at 980 Mbps and 64-bytes long packets at a rate of 500 Mbps. In both experiments the IXP1200-based splitter was able to receive the traffic without any packet loss.

C. Evaluation of Locality Buffers

1) Environment: To evaluate the effect of Locality Buffers we use an experimental environment consisting of one traffic source, one IXP1200-based traffic splitter, and two sensors running the snort Intrusion Detection System as shown in Figure 18. The traffic source is a

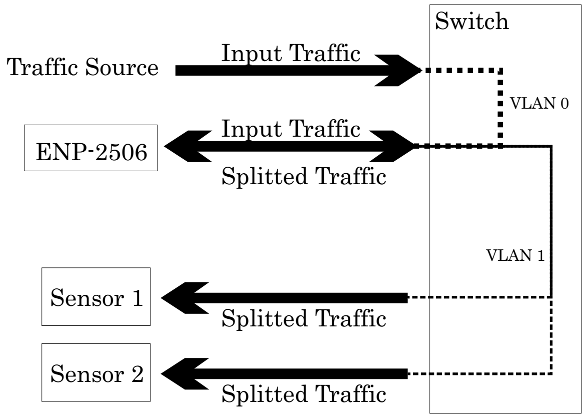

Fig. 18. Experimental Evaluation System for the IXP1200-based Splitter

personal computer equipped with a dual processor AMD running at 2 GHz with 512 KB cache, 512 MB of main memory, one 64-bit Gigabit Ethernet PCI interface, and a SCSI-based hard disk subsystem. The traffic source, running Linux version 9 with kernel 2.4.20, replays the `nlanr.MRA.1031627450` trace file.

Each sensor is a personal computer equipped with a Pentium 4 processor clocked at 2 GHz, 512 KB of cache, 1 GB of main memory, and runs the *snort* Intrusion Detection System version 2, using the default set of signatures and no preprocessors. The operating system of each sensor is Linux version 8.0 with kernel version 2.4.20.

The IXP Splitter running on the ENP-2506 board is configured with 21 locality buffers. Each one of five locality buffers was dedicated to one of the following categories of traffic as follows:

- one buffer for all packets destined to port 80 (web client traffic)
- one buffer for all packets originating from 80 (web server traffic)
- one buffer for all packets destined to port 119 (nntp traffic)
- one buffer for all packets destined to port 1214 (Kazaa traffic)
- one buffer for all packets destined to port 6699 (Napster traffic)

The rest of the buffers were used to accommodate packets by hashing on the destination port of each incoming packet. Each locality buffer was large enough to accommodate about 300 packets of 512 Bytes each.

During each experiment we measured user time plus kernel time for each of the sensors.

TABLE V

PERFORMANCE EVALUATION OF LOCALITY BUFFERS ON THE IXP1200-BASED SPLITTER

	Aggregate Time (seconds)	Slowest Sensor (seconds)
without locality buffers	31.8	16.4
using locality buffers	27.0	14.1
percentage difference	17.7%	16.3%

2) Results: Table V summarizes the measurements obtained for two sensors. The second row presents the performance of the system that does not use locality buffers. The third row presents the performance of the system that uses locality buffers. The second column presents the aggregate time of the two sensors and the third column presents the time of the slowest sensor. We see that the use of locality buffers improves the aggregate time (second column) by as much as 17%. Locality buffers also improve the running time of the slowest sensor by 16% as well. These results are consistent with those reported in Figure 4. Note that the aggregate time report in Figure 4 is close to 47 seconds, while the aggregate time reported in Table V is close to 32 seconds. This difference can be attributed to the different hardware the sensors were running on top of. Indeed, in Figure 4 we use processors clocked at 1.13 GHz, while the results reported in Table V are based on processors clocked at 2 GHz. In both cases, however, we see that locality buffers improve the performance of the overall Intrusion Detection System.

VII. SUMMARY AND FUTURE WORK

In this paper we have proposed an active traffic splitter architecture for Intrusion Detection. Rather than acting as a passive load balancing component, we argue that the traffic splitter should actively manipulate the traffic stream in a way that increases sensor performance.

We have presented and analyzed two specific examples of performance-enhancing mechanisms. The first is *early filtering*, where a subset of the traffic is processed on the traffic splitter and filtered out in order to reduce the load on the sensors. In its most simple form, early filtering increases sensor performance by 8% by filtering out roughly 32% of the packets that are not subject to content matching. The header rules that constitute early filtering are only a small fraction of the nIDS ruleset making it easy to implement on the traffic splitter. The second method is *locality buffering*, where packets classified to the same subset of nIDS rules are buffered together before being forwarded to the sensors. By grouping same-type packets and sending them to the sensor back-to-back, this method increases memory access locality on the nIDS sensors resulting in improved performance. We have examined the effect of different buffering policies and buffer parameters and our results indicate that using 32 locality buffers of 256 KB each and a policy of using dedicated buffers for the major traffic groups results in a 10% reduction in nIDS load. When using both methods together, overall system performance is improved by 19.8%, while the running time of the most loaded sensor is improved by 14.4%.

We have implemented the traffic splitter with locality buffering on top of an IXP1200 network processor. Our results show that the traffic splitter is capable of handling traffic as high and 500 Mbps for minimum-size packets. Moreover, our experimental results suggest that locality buffering improves the performance of Intrusion Detection based on a network-processor-based splitter by as much as 17%.

Based on our results we believe that network-processor-based traffic splitters are an effective way to scale the performance of Intrusion Detection Systems, enabling them to effectively monitor the increasingly growing Internet links.

REFERENCES

[1] K. G. Anagnostakis, S. Antonatos, M. Polychronakis, and E. P. Markatos, "E^2xB: A domain-specific string matching algorithm for intrusion detection," in *Proceedings of IFIP International Information Security Conference (SEC'03)*, May 2003.

[2] S. Antonatos, K. G. Anagnostakis, M. Polychronakis, and E. P. Markatos, "Benchmarking and design of string matching intrusion detection systems," ICS-FORTH, Tech. Rep. 315, December 2002.

[3] R. Bace and P. Mell, *Intrusion Detection Systems*. National Institute of Standards and Technology (NIST), Special Publication 800-31, 2001.

[4] Z. Cao, Z. Wang, and E. W. Zegura, "Performance of hashing-based schemes for internet load balancing," in *Proceedings of IEEE Infocom*, 2000, pp. 323–341.

[5] C. J. Coit, S. Staniford, and J. McAlerney, "Towards faster pattern matching for intrusion detection, or exceeding the speed of snort," in *Proceedings of the 2nd DARPA Information Survivability Conference and Exposition (DISCEX II)*, June 2002.

[6] M. Fisk and G. Varghese, "An analysis of fast string matching applied to content-based forwarding and intrusion detection," University of California - San Diego, Tech. Rep. CS2001-0670 (updated version), 2002.

[7] G. Goldszmidt and G. Hunt, "Scaling internet services by dynamic allocation of connections," in *Proceedings of the Sixth IFIP/IEEE International Symposium on Intergrated Network Management*, May 1999, pp. 171–184.

[8] E. Johnson and A. Kunze, *IXP1200 Programming*. Intel Press, 2002.

[9] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer, “Stateful Intrusion Detection for High-Speed Networks,” in *Proceedings of the IEEE Symposium on Research on Security and Privacy*. Oakland, CA: IEEE Press, May 2002.

[10] E. P. Markatos, M. D. Flouris, D. N. Pnevmatikatos, and M. G. H. Katevenis, “Web-conscious storage management for web proxies,” Institute of Computer Science, Foundation of Research and Technology Hellas, Tech. Rep. 275, 2000. [Online]. Available: citeseer.nj.nec.com/328247.html

[11] S. McCanne, C. Leres, and V. Jacobson, “libpcap,” 1994, lawrence Berkeley Laboratory, Berkeley, CA, available via anonymous ftp to *ftp.ee.lbl.gov*.

[12] NLANR, “MRA traffic archive,” September 2002, *http://pma.nlanr.net/PMA/Sites/MRA.html*.

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

[14] M. Roesch, “Snort: Lightweight intrusion detection for networks,” in *Proc. of the 1997 USENIX Systems Administration Conference (LISA)*, November 1999, (software available from *http://www.snort.org/*).

[15] TopLayer, “IDS load balancer,” product description available through *http://www.toplayer.com/*.