Computer Science Department
University of Crete

Network Intrusion Prevention on Multilevel Processing Architectures

Master's Thesis

Konstantinos Xinidis

4th November 2004
Heraklion, Greece

Network Intrusion Prevention on Multilevel Processing Architectures

by

Konstantinos Xinidis

Master's Thesis

Department of Computer Science
University of Crete

Abstract

Network intrusion prevention systems provide an important proactive defense capability against security threats by detecting and blocking network attacks. This task can be highly complex, and software-based network intrusion prevention systems are currently not capable of handling high speed links.

This work focuses on the design and implementation of a high-performance, low-cost, flexible, and scalable network intrusion prevention system that combines software-based network intrusion detection engines and a network processor board. The network processor acts as a customized load balancer that cooperates with a set of content-based network intrusion detection engines in processing network traffic. We show that the components of such a system, if designed properly, can achieve high performance, by eliminating redundant processing and communication.

We describe a system architecture and present a prototype built using low-cost, off-the-shelf technology: an IXP1200 network processor evaluation board and commodity PCs. Our evaluation shows that our enhancements reduce the processing load of the network intrusion detection engines by at least 45%. The result is a system that can handle a fully-loaded Gigabit Ethernet link using at most four detection engines.

Keywords: intrusion detection, intrusion prevention, load balancing, network processors

Thesis Supervisor: Evangelos P. Markatos

Title: Associate Professor

Ιεραρχικές Αρχιτεκτονικές από Επεξεργαστές για Αποτροπή Εισβολέων σε Δίκτυα.

Κωνσταντίνος Ξυνίδης

Μεταπτυχιακή Εργασία

Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

Περίληψη

Τα συστήματα αποτροπής εισβολέων σε δίκτυα παρέχουν την ικανότητα της πρόληψης και αμυντικής θωράκισης ενάντια σε απειλές ασφάλειας με το να ανιχνεύουν και να παρεμποδίζουν επιθέσεις που προέρχονται από το δίκτυο. Αυτό το έργο μπορεί να γίνει πολύ περίπλοκο, με αποτέλεσμα τα συστήματα αποτροπής εισβολέων σε δίκτυα που χρησιμοποιούν λογισμικό να αδυνατούν, προς το παρόν, να προστατεύσουν δίκτυα υψηλών ταχυτήτων.

Η εργασία αυτή ασχολείται με τον σχεδιασμό και την υλοποίηση ενός συστήματος αποτροπής εισβολέων σε δίκτυα, που χαρακτηρίζεται από υψήλες επιδόσεις, μικρό κόστος, ευελιξία και κλιμάκωση. Το σύστημα αυτό χρησιμοποιεί μηχανές ανίχνευσης εισβολέων υλοποιημένες σε λογισμικό και έναν επεξεργαστή δικτύου. Ο επεξεργαστής δικτύου συμπεριφέρεται σαν ένας ειδικά προσαρμοσμένος κατανεμητής φόρτου ο οποίος λειτουργεί σε συνεργασία με ένα σύνολο από μηχανές ανίχνευσης εισβολέων για να ελέγχει την κίνηση που διέρχεται απο το δίκτυο. Δείχνουμε ότι τα επιμέρους εξαρτήματα ενός τέτοιου συστήματος, εάν έχουν σχεδιαστεί κατάλληλα, μπορούν να επιτύχουν υψηλές επιδόσεις με την εξάλειψη της πλεονάζουσας επεξεργασίας και επικοινωνίας.

Περιγράφουμε την αρχιτεκτονική ενός συστήματος και παρουσιάζουμε ένα πρωτότυπο σύστημα που υλοποιήθηκε χρησιμοποιώντας χαμηλού κόστους και εύκολα προσβάσιμη τεχνολογία: έναν επεξεργαστή δικτύου που ονομάζεται *IXP1200* και οικονομικά PCs. Η αξιολόγηση του συστήματος δείχνει ότι οι βελτιστοποιήσεις μας μειώνουν τον φόρτο των μηχανών ανίχνευσης εισβολέων τουλάχιστον κατά 45%. Το αποτέλεσμα είναι ένα σύστημα που μπορεί να προστατεύσει ένα πλήρως φορτωμένο

Gigabit δίκτυο χρησιμοποιώντας το πολύ τέσσερις μηχανές ανίχνευσης εισβολέων.

Επόπτης Μεταπτυχιακής Εργασίας: Ευάγγελος Π. Μαρκάτος

Acknowledgments

This work was financially supported by the Institute of Computer Science (ICS) of the the Foundation for Research and Technology - Hellas (FORTH), Heraklion, Greece. Besides ICS-FORTH, i would also like to thank all those people who helped me thoughout this work. First of all, i would like to thank my supervisor, Evangelos P. Markatos who gave me the possibility to work on the subject of network intrusion detection and prevention. I would also like to thank him for his guidance and support thoughout this work. Our discussions helped me understand how brilliant people are thinking and how a problem must be handled in order to be solved. His simple, yet scientific, way of thinking became valuable guide for me during the last few years.

In addition to prof. Evangelos P. Markatos, another person who helped me very much and i was very lucky to work with, is Konstantinos Anagnostakis. I truly believe that Konstantinos is the ideal co-worker. I would like to thank him for helping me stand on my foot when i was desperate during hard to solve problems. Also, his suggestions and observations proved invaluable.

I would also like to thank the rest of the Advanced Computing Systems group at ICS-FORTH for their help. Especially, i would like to thank Manolis Ploumidis for his help on implementing the packet caching technique on the *IXP1200*. Also, i would like to thank Spiros Antonatos, Mihalis Polihronakis, Olga Dokianaki, Thalia Vasiliadou and John Yannakopoulos for their corrections on parts of this thesis report.

I also thank my brothers, my parents and my grand-parents for their love and support all these years: nothing would have been accomplished without their contribution. Thanks are also to my friend Konstantinos Dimitriou for showing me the right way of thinking. His unbelievable way of solving problems showed me that there are people who are extraordinary clever and no matter how much we try, we cannot reach them. I therefore dedicate this work to him.

Στον φίλο μου Κωνσταντίνο Δημητρίου

Κι αν έφυγες απ᾽ την ζωή

πάντα θα σε θυμάμαι

σαν ένα άστρο που ᾽λαμπε

μα έπεσενε χάμε.

Contents

List of Figures

List of Tables

1 Introduction

The increasing importance of network infrastructure and services along with the high cost and difficulty of designing and enforcing end-system security policies has resulted in growing interest in complementary, network-level security mechanisms, as provided by firewalls and network intrusion detection and prevention systems.

High-performance firewalls are rather easy to scale up to current edge-network speeds because their operation involves relatively simple operations, e.g., matching a set of Access Control List-type policy rules against fixed-size packet headers. Unlike firewalls, network intrusion prevention systems (NIPSes) are significantly more complex and, as a result, are lagging behind routers and firewalls in the technology curve. The complexity stems mainly from the need to analyze not just packet headers but also packet content and higher-level protocols. Moreover, the function of NIPSes needs to be updated with new detection components and heuristics, considering the progress in detection technology as well as the continuously evolving nature of network attacks.

Both complexity and the need for flexibility make it hard to design high-performance NIPSes. Application-Specific Integrated Circuits (ASICs) lack the needed flexibility, while software-based systems are inherently limited in terms of performance. One design that offers both flexibility and performance is the use of multiple software-based systems behind a hardware-based load balancer. Although such a design can scale up to edge-network speeds, it still requires significant resources, in terms of the number of software-based systems, the required rack-space, etc. It is therefore important to consider ways of improving the performance of such systems.

This thesis explores the role that high-speed network processors (NPs) can play in scaling up network intrusion prevention systems. We focus on ways of exploiting the performance and programmability of NPs for boosting network intrusion prevention. We describe the architecture of a high-performance, low-cost, flexible, and scalable NIPS that is composed of network processors and general purpose processors. We present the allocation of operations to components and the trade-offs we faced during designing and prototyping the system.

1.1 Design Challenges

We faced a number of design challenges in constructing our system with respect to performance, flexibility, and scalability:

1.1.1 High-Performance

The primary metric of interest in the design of a NIPS is throughput. The goal here is to be able to operate at network speeds of at least 1 Gbit/s without packet losses. We assume that we cannot tolerate an undetected attack; therefore, the system must be capable of analyzing all the incoming traffic under the most stringent conditions.

A second important performance goal is minimizing the latency induced by the NIPS. There is a direct relationship between latency introduced by a networking device and the maximum throughput of TCP connections [1]. If the NIPS is to be used at the boundary between an enterprise network and the Internet, latencies in the order of a few milliseconds may be tolerable. If the NIPS is deployed internally, and the network needs to support high-bandwidth local services (such as file sharing, etc.) the latency

[1] Recall that $Throughput = \frac{Window}{RTT}$ where *Window* is the maximum TCP window size (default value is 64 Kbytes) and *RTT* is the round trip time in the network.

requirements are even more stringent.

Particularly, there is a critical value for the round trip time (RTT) of a packet in each network. If the latency is below this critical value, TCP throughput is unaffected – it is the line speed of the underlying network which becomes the bottleneck – above this critical value, however, TCP throughput is negatively impacted. The critical value for RTT in a network supporting Gigabit speeds is 0.5 milliseconds. Thus, if we want the throughput of TCP to be unaffected, we must ensure that the imported latency of our NIPS is smaller than 0.5 milliseconds. However, Gigabit Ethernet links, rarely carry only a single TCP connection. Rather, a Gigabit Ethernet link supports hundreds, if not thousands of TCP connections, and this multiplexing mitigates the impact of latency on the overall throughput of the link [16]. In other words, it is possible to import latency greater than 0.5 milliseconds without affecting the throughput of a link due to the high number of TCP connections.

1.1.2 Flexibility and Scalablity

A NIPS needs to be flexible and scalable, both for scaling up to higher link speeds and more expensive detection functions, as well as for updating the detection heuristics. If the protection of a faster link or a more fine-grained detection is required, it would be desirable to reuse as much as possible of the existing hardware. Clearly, this property does not hold for ASIC-based NIPSes. Besides, it is remarkable that almost all NIPSes providers ignore this dimension [15, 28, 29, 43]. Furthermore, notice that a prerequisite of flexibility is simplicity as extending a complex system may be hard and error-prone. It is therefore desirable for the hard-to-program elements of our system to be as generic as possible.

2
Background

In this Chapter we present a summary of the important concepts that this study is based on. We first present the differences between NIPSes, NIDSes and firewalls. Then, we give an introduction to network intrusion prevention systems and to possible implementations of a NIPS. Subsequently, follows a brief description of the architectural characteristics of network processors. Finally, we briefly discuss load balancing and present the requirements that load balancing for network intrusion prevention must satisfy.

2.1 Differences Between NIPSes, NIDSes and Firewalls

It is important not to confuse network intrusion detection (monitoring) with network intrusion prevention. NIDSes detect attacks and provide information about the attacks so that an administrator can perform an inspection and determine if something is not as it ought to be. Since intrusion detection is a passive technology, it can tolerate detection techniques that are not perfect. The worst case scenario is that you have false positives. Network intrusion detection is a passive detection technology, utilizing broad detection methods sufficient to understand and characterize the traffic that is present on a network. Network intrusion prevention, on the other hand, is quite different. Its job is to proactively prevent attacks from entering your network opposed to just detecting those attacks. A NIDS operates beside the network, offline, and analyzes the traffic as it goes by. On the contrary, a NIPS operates inline, inspecting all packets going inbound or outbound. With intrusion prevention, only precise detection/prevention methods can be used to insure that there are little to no false positives and that legitimate traffic is not mistakenly blocked. It is important to understand that both NIDSes and NIPSes have an important role in a network infrastructure. NIPSes are meant to minimize the risk of known attacks and anomalies that can only cause damage to your network. Where a NIDS gives you insight into your networks patterns and behaviors and helps you to gather forensic data over time.

Finally, firewalls on the other hand, while operating inline, just like a NIPS, they do not inspect the payload of the packets. They merely check the headers of the packets and block traffic not destined to a list of permitted ports. Although this is useful in many cases, firewalls are not capable of blocking the biggest part of the known attacks.

2.2 Introduction to Network Intrusion Prevention Systems

2.2.1 What is an Intrusion?

An intrusion is “the act of thrusting in, or of entering into a place or state without invitation, right, or welcome”[45]. In the context of computer systems, intrusion is any unauthorized action in an attempt to compromise a system. Figure 2.1 presents a typical intrusion scenario. For an attacker, the first step is to select the system to attack, which is also known as *outside reconnaisance*. After selecting his victim, the attacker proceeds with *inside reconnaisance*. Particularly, in this step the attacker gathers information about the victim which will help him find possible vulnerabilities or the victim’s Achilles’ heel. Then, the attacker takes advantage of these vulnerabilities using some exploits and compromises the system. After that, the attacker may have a plethora of options that ranges from stealing confidential information to render unusable the compromised system (e.g erasing hard disks).

2.2.2 What is a Network Intrusion Prevention System?

Network intrusion detection and prevention systems analyze information about the activities performed in a network, looking for evidence of malicious behavior, and block the offending traffic auto-

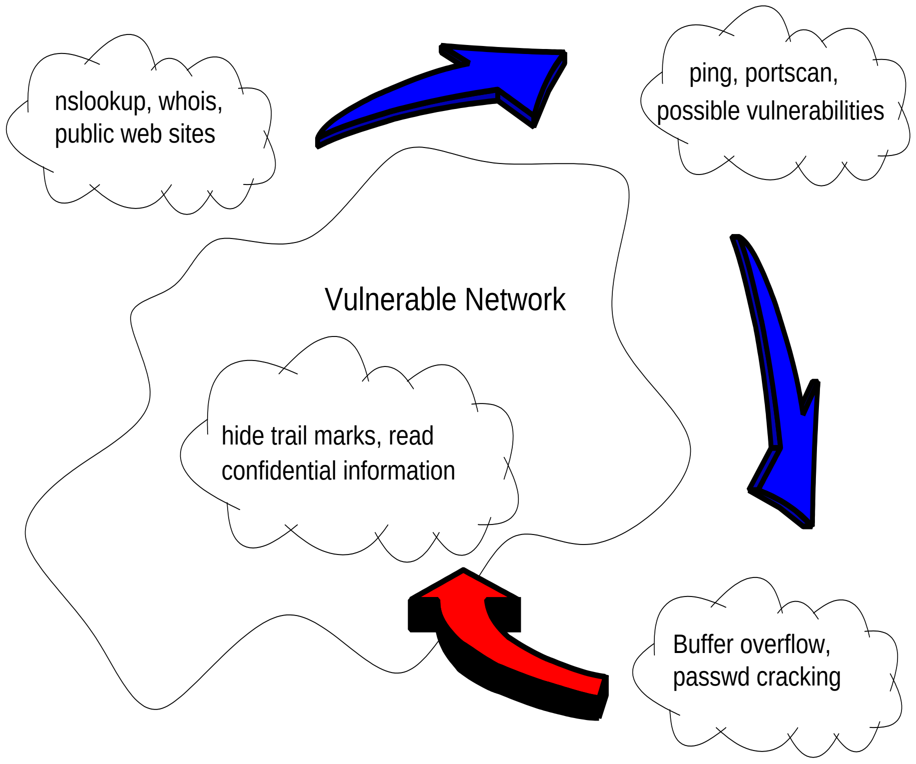


FIGURE 2.1: A Typical Intrusion Scenario.

matically before it does any damage. The information comes in the form of raw network traffic obtained by monitoring a network link. The collected data are used by NIPSes in two different ways, according to two different approaches: *anomaly detection* and *misuse detection* systems [20].

Anomaly detection systems collect historical data about the activity of a system. Then, given some specifications of the normal behavior of the system, a profile representing the normal operation of the system is constructed. The specifications, for example, may contain the state of the network's traffic load, breakdown, protocol, and typical packet size. During detection, the NIPS tries to identify patterns of activity that deviates from the defined profile (anomalous activity). This approach is based on the notion that attacks tend to look different in some fashion than legitimate computer use. Rather, misuse detection systems take a complementary approach. They are equipped with a number of attack descriptions (or signatures) that are matched against the stream of network data, looking for evidence that a known attack is occurring. Essentially, a misuse detection system looks for a specific attack that has already been documented.

Each of these approaches has its pros and cons. Misuse detection systems can perform detailed analysis of the network data and they usually produce only a few false positives, but they can't detect novel attacks. On the other hand, anomaly detection systems have the advantage of being able to detect previously unknown attacks. This advantage is paid for, in terms of the large number of false positives and the difficulty of training a system with respect to a very dynamic environment.

In this thesis, we concentrate on misuse detection NIPSes. We decided to use misuse detection NIPSes because they are more mature than the anomaly detection NIPSes. In particular, as we have already mentioned, they generate less false positives, and are well understood.

2.2.3 Basic Functions of a Network Intrusion Prevention System

The functionality of a common NIPS can be divided into three different phases: (1) the protocol decoding phase, (2) the detection phase, and (3) the prevention phase. In the first phase, the raw packet stream is seperated into connections representing end-to-end activity of hosts. A connection, in case of IP traffic, can be identified by the source and destination IP addresses, transport protocol and UDP/TCP ports. Then, a number of protocol-based operations are applied to these connections. The protocol handling ranges from network layer to application layer protocols. Some of the operations applied by the protocol-based handling are IP defragmentation, TCP stream reconstruction, identification of the URI in HTTP requests etc. The second phase consists of the actual detection. Here, the packet (or an equivalent higher-level protocol data unit) is checked against a database of detection heuristics representing attack patterns. Then follows the prevention phase. The action of this phase depends on the result of the previous one. If no attack is found, the packets are forwarded to their destination. If malicious activity is observed, then the prevention engine blocks the suspicious traffic by not forwarding the packets belonging to the offending connection(s). Other representative actions of the prevention engine include the rejection of connections by the offending source host for a period of time, or the logging of the offending connections.

2.2.4 Protecting the Network Infrastructure

The reason behind the existence of a NIPS is to protect the network infrastracture. However, as networks evolve, the data volume that a NIPS must process increases. There are two approaches to

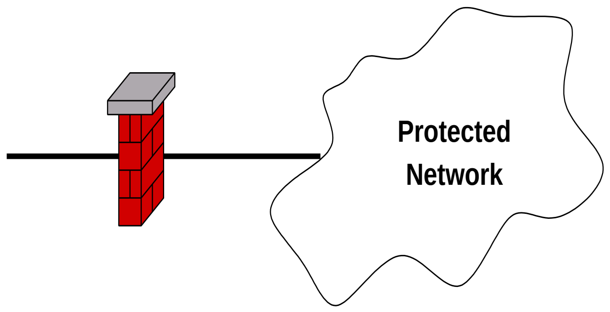


FIGURE 2.2: Single Location Placement.

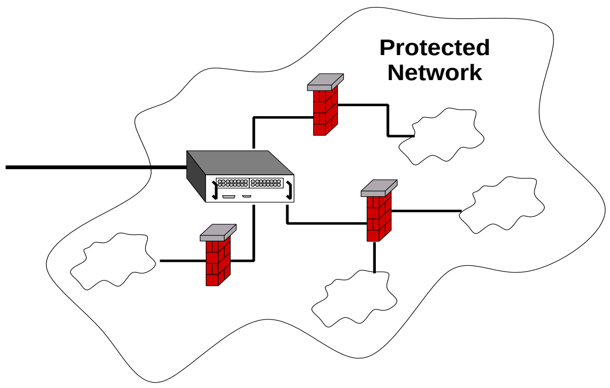


FIGURE 2.3: Multiple Locations Placement.

analyze this amount of data in real-time: use powerful sensors, [1] or use multiple sensors at the network periphery where the traffic volume is lower [20].

In the first approach (Figure 2.2), the whole traffic is captured at a single location by a muscular NIPS. Obviously, the difficulty with this technique is how to implement such powerful NIPSes. The second approach (Figure 2.3) incorporates the deployment of multiple sensors at the network periphery, close to the hosts which the system must protect. This technique takes advantage of the fact that by moving the inspection to the periphery of the network, a natural partitioning of traffic will occur. Unfortunately, numerous problems stem from this approach. First of all, it is cumbersome to deploy and manage a highly distributed set of sensors. Second, correct sensor positioning can be a challenging task due to the dynamic nature of networks. Last but not least, this approach is unable to provide an integrated, ”big picture” view of the network security status. Particularly, attacks that might appear irrelevant in the

[1]In this thesis the terms NIPS and sensor are used interchangeably.

context of a single host, might be extremely dangerous when are considered across the network. For all these reasons, we decided to follow the first approach and try to tackle the performance problem that this approach holds.

2.3 A Design Space for Network Intrusion Prevention Systems

Here, we are going to present all the possible network intrusion prevention system architectures and discuss their advantages and disadvantages. To start with, we can categorize the architectures of a NIPS regarding the type of hardware used. In particular four main categories exist: (1) software-based NIPSes implemented on General Purpose Processors (GPPs), (2) software-based NIPSes implemented on network processors, (3) hardware-based NIPSes, frequently implemented using Application Specific Intergarted Circuits (ASICs) or Field Programmable Gate Arrays (FPGAs) [2] and (4) hybrid NIPSes that consist of software- and hardware-based components. Before presenting the characteristics of these four main categories we give a short definition of some keywords that we will use extensively.

2.3.1 Useful Definitions

Because high-performance, flexibility, and scalability are key elements of this thesis, they are defined here.

High-Performance

A high-performance NIPS is one that is capable of analyzing all the traffic transfered by high-speed network links. To accomplish this goal, a high-performace NIPS must use the available processing capacity as efficiently as possible. It must minimize redundant processing and communication, effectively using the available resources for analyzing the traffic in order to identify intrusion attempts.

Flexibility

A NIPS is flexible, if it permits the operator, where the ”operator” is someone other than the manufacturer, to put new functionality or upgrade the existing one with minimal effort and cost. For instance, software-based NIPSes are flexible by their nature while hardware-based NIPSes require significant efforts in order to add new functionality.

[2] A device that can be reprogrammed at the gate level.

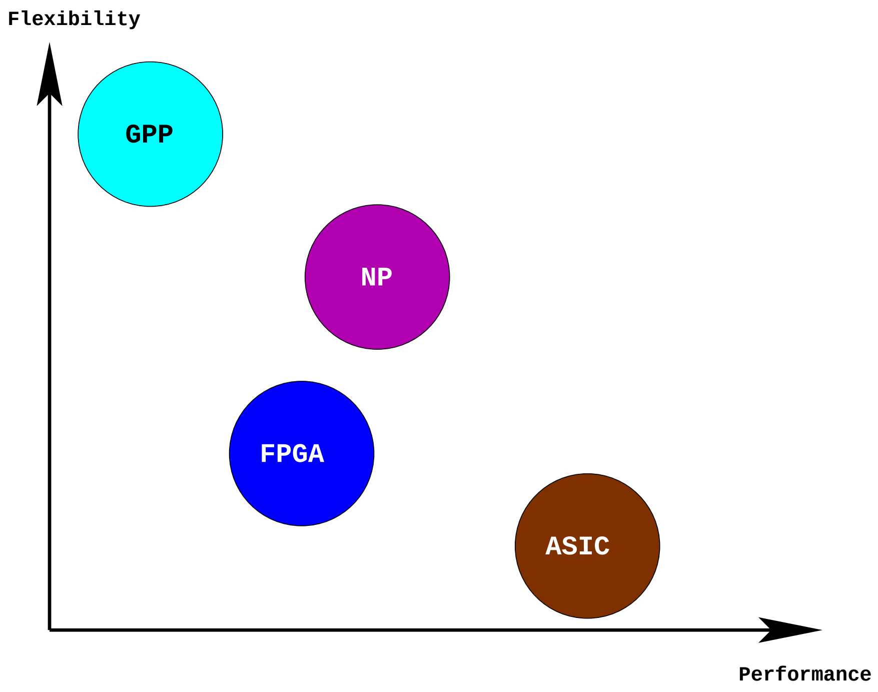


FIGURE 2.4: Possible Implementations of a NIPS.

Scalability

A NIPS is scalable, if it is capable of scaling up to higher link speeds. If the protection of a faster link or a more fine-grained detection is required, it would be desirable to reuse as much as possible of the existing infrastructure.

2.3.2 Possible Implementations of a NIPS

In Figure 2.4 we present the first three categories regarding performance and flexibility. As we observe, while GPPs provide the most flexible solution, they have relatively poor performance. On the other hand, ASICs have impressive performance characteristics but provide limited flexibility. Somewhere in the middle are the NPs and the FPGAs. An NP is more flexible than an ASIC but is less powerful. Lastly, FPGAs are slower and less flexible compared to NPs. Regarding the hybrid NIPSes, many possible architectures exist and as a result their characteristics vary.

Table 2.1 presents all possible implementations of a NIPS and the characteristics of each implementation. As we observe, there are many possible combinations of the three basic categories. In the next paragraphs we give a short description of each combination and present its advantages and disadvan-

NIPS Impl.	Flexibility	Performance	Cost per Part
ASIC	🟡	⚫	🟢
FPGA	🔴	🟢	⚫
NP	🔵	🔴	🔵
GPP	⚫	🟡	🟡
ASIC/GPP	🔴	🔴	🟡
FPGA/GPP	🔴	🔴	🔴
NP/GPP	🔵	🔴	🟢
ASIC/NP	🟢	🔵	🔴
FPGA/NP	🟢	🔵	🔵

TABLE 2.1: Characteristis of Possible Implementations of a NIPS.

ASIC	Application-Specific Integrated Circuit
FPGA	Field Programmable Gate Array
NP	Network Processor
GPP	General Purpose Processor

🟡 Lowest 🟢 Low 🔴 Medium 🔵 High ⚫ Highest

tages.

ASIC As we have mentioned before, an ASIC has the best possible performance but it is not flexible. Moreover, the cost for the manufacturer to produce a unit if it has completed the implementation is small. However, the cost to design hardware is high. On the other hand, the cost for the customer to buy a hardware NIPS is very high and in many cases prohibitive. Thus, the conclusion is that it is expensive both for a manufacturer to design a hardware NIPS and a customer to buy such a NIPS.

FPGA An FPGA, has low performance, medium flexibility, and exhibits a high cost both for the manufacturer and the customer. Additionally, the cost to design a NIPS in hardware is high and as a result this solution has not been adopted by any company. Currently, there is research in constructing NIPSes using an FPGA-based approach [27, 24, 9].

NP NPs are a hybrid approach between ASICs and GPPs. They provide good performance without sacrifizing flexibility. They are especially tuned for network-based applications, such as a NIPS, and for this reason they are an attractive solution. The cost to buy an NP is relatively high due to the fact that is constructed in low volumes. However, if the NPs are adopted for general network applications their cost will be almost the same as GPPs. Regarding the cost to develop an application on an NP, it is much lower than in the ASIC case but is higher than in the GPP case desribed next. Moreover, note that while NPs exhibit a good performance, this performance is related to operations that are common to network applications such as packet forwarding, packet classification etc. However, NPs are not usualy optimized for performing operations such as string searching in the payload of the packets and other such processor-intensive operations that are present in most NIPSes. In other words, an NP is more effective on performing data-intensive operations. Thus, NPs are commonly used in conjuction with either an ASIC/FPGA or a GPP to implement a fully-functional NIPS.

GPP GPPs exhibit the highest possible flexibility and lowest performance. Note that, when we refer that a GPP has low performance, we mean low performance regarding operations that are frequently performed in a network application. In other operations, such as string searching, GPPs have better performance than other solutions like NPs. Unfortunately, a software-based NIPS implemented on a GPP is not capable of monitoring a Gigabit link. However, for lower speed links it is the ideal solution. It is cheap to buy and cheap (and fast) to develop an application on a GPP due to the vast number of existing tools.

ASIC/GPP-FPGA/GPP There are NIPSes that try to combine the performance of ASICs with the flexibility of GPPs. These configurations are a good solution that provides high performance and relative good flexibility. However, the presence of the ASIC, provide an upper bound on the flexibility of such a system. In such configurations, probably, the ASIC is responsible for forwarding the packets while the GPPs are executing the detection heuristics. The FPGA/GPP solution is similar with the ASIC/GPP solution but it costs more money and has poorer performance. Thus, it is not adopted by anyone as far as we know.

NP/GPP This solution combines the performance and flexibility of NPs with the flexibility of GPPs to build a NIPS with the best tradeoff between performance and flexibility. Unlike the ASIC/GPP case, the presence of the NP does not limit the flexibility of the system while the cost is acceptable, provided that a small number of NPs is used. In such a configuration, the NP is responsible for performing packet forwarding while the GPPs perform the processor intensive task of packet inspection.

ASIC/NP-FPGA/NP The combination of ASICs or FPGAs with NPs provide high performance and little flexibility. In such a configuration the processing intensive task of packet inspection can be accomplished in hardware, while the NP is busy forwarding packets or performing operations such as IP defragmentation and TCP stream reconstruction.

2.4 Introduction to Network Processors

A network processor is a programmable silicon device that is tailored to efficiently process packets. The goal of network processors is ambitious: combine the speed of custom hardware with the flexibility and low development cost of software-based systems. A network processor represents a complex combination of conventional processing, special-purpose hardware, and replicated units. In the next sections we present four major aspects of network processor architectures: approaches to parallel processing, elements of special purpose hardware, structure of memory architectures, and on-chip communication mechanisms [30, 10].

2.4.1 Parallel Processing

In order network processors to be useful, they must be able to meet the increasing line speed requirements of today's networks. To accomplish this, network processors have taken advantage of the inherent parallelism in various networking algorithms. Parallelism is exploited at three different levels: processing element level, instruction level and word/bit level.

Processing Element Level

We define a processing element (PE) to be a processor that decodes its own instruction stream. Given the data parallelism present in most packet processing systems, the majority of network processors employ multiple PEs to take advantage of this parallelism. Those NPs can be categorized into two prevalent configurations: pipelined and symmetric.

In the pipelined approach, each processor is designed for a particular packet processing task. Packets flow through PEs – once a PE is finished processing a packet, it hands of the packet to the next PE. On one hand, these architectures restrict the communication between programs on different PEs, and as a consequence, are easier to program. On the other hand, there are strict timing requirements to be met by each program running on every PE.

In the symmetric approach, each PE is able to perform similar functionality. All PEs are usually programmed to perform similar functionality. These network processors are more flexible but are difficult to program. The *IXP1200* is an example of this type of architecture.

Instruction Level

Some network processors issue multiple instructions per cycle per PE. However, the benefits are not warranted due to the observation that most networking applications do not have the required instruction level parallelism.

Bit Level

This type of parallelism depends mainly on the application. The data types used by the application and the operations performed on these data types play a central role in the applicability of this technique. For example, many network processors have circuitry to efficiently compute the CRC field of a packet header.

2.4.2 Special Purpose Hardware

Towards the goal of meeting increasing network processing demands, network processor architects decided to implement commonly used functions in hardware instead of having a slower implementation using an ordinary ALU. However, a trade-off exists between the applicability of the hardware and the speedup obtained. The two categories of special-purpose hardware used are co-processors and special functional units.

Co-Processors

A co-processor is characterized by the lack of an instruction decode unit. As a consequence, a co-processor must be triggered by another PE. After the work is assigned by a PE to the co-processor, the results are computed asynchronously. Operations ideally suited for co-processor implementation are well defined, expensive and/or cumbersome to execute within an instruction set, and prohibitively expensive to implement as an independent special functional unit. An example of a co-processor is the hash engine of the *IXP1200* network processor.

Special Functional Units

A special functional unit, in contrast to a co-processor, computes a result within the pipeline stage of a PE. Operations well suited for hardware implementation are cumbersome and error-prone to implement in software, yet very easy to implement in hardware. For instance, NPs have special functional units for common networking operations like pattern matching and bit manipulation.

2.4.3 Structure of Memory Architectures

Another feature of network processors is the structure of memory architectures. The major tactics that NPs used are multi-threading, memory management and task-specific memories.

The majority of network processors do not require the use of an operating system (OS) running on top of them. In reality, the overhead of an operating system is prohibitively expensive for a processor designed for data-plane processing. To mitigate the lack of an OS, network processors include more hardware support for common OS functions, like multi-threading and memory management.

In order to use a network processor efficiently, memory access latency must be hidden in some way. The most broadly used approach to hide memory latency is multi-threading. Multi-threading allows the PE to be used to process other streams while another thread waits for a memory access to be completed. Due to the high cost of implementing multi-threading in software, many network processor designers provide hardware support for multi-threading. Specifically, many NPs have separate register banks for different threads, and hardware units to schedule and swap threads with zero overhead.

Concerning memory management, some network processors have hardware support for queues that can be used as free lists, thus obviating the need for a separate OS service routine. Other NPs offer special circuitry that handles the common I/O path. Finally, task-specific memories are blocks of memory coupled with some logic for specific storage applications. Examples of such memories are the Content Addressable Memories (CAMs) which are used mainly for classification.

2.4.4 On-Chip Communication Mechanisms

Generally, on-chip communication mechanisms depend on the processing element configuration. In the pipelined approach discussed above, most communication architectures are point-to-point, between PEs, memory, and co-processors. In the symmetric approach, PEs have full connectivity with multiple buses.

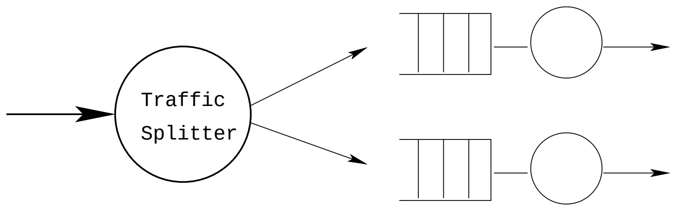


FIGURE 2.5: A Load Balancing System.

2.5 Introduction to Load Balancing

Load balancing (also known as load sharing) is a key technique for improving the performance and scalability of a system [6]. A load balancing system typically comprises of a traffic splitter and multiple outgoing links, as shown in Figure 2.5. In such a system, the traffic splitter receives an incoming packet from a higher-speed link and forwards it to one of the lower-speed outgoing links. A good load balancing system should be able to split the traffic to the multiple outgoing links evenly, or by some pre-defined proportion.

In case of a NIPS, the load balancing splitter is usually used to distribute the traffic to multiple intrusion detection/prevention engines in order to tackle the performance problem that we have already discussed. Subsequently, we describe the requirements of a load balancing algorithm in order to be used by a NIPS.

2.5.1 Requirements

There is a number of basic requirements that traffic splitting schemes should meet for network intrusion prevention systems load balancing:

Low Overhead

Traffic splitting is executed for every packet in the packet forwarding path, thus the per-packet overhead it introduces is of major concern. Traffic splitting algorithms should be very simple and preferably keep no or little state regarding the mapping of packets to outgoing links.

High Efficiency

Poor traffic distribution will result in uneven utilization of the detection engines and possible dropping of packets. Even if an attack is not missed, the latency imported on the packets will be increased resulting in possible TCP implications. Thus, a traffic splitter should try to distribute traffic as evenly as possible.

Flow Preserving

In order to understand this requirement, we first have to explain what is a *flow*. In principle, we define that two or more packets belong to the same flow, when they share some common attributes. For example, we may define that all the packets that have the same payload belong to the same flow (for example the packets assembling an Internet worm), or all the packets with the same source IP address (all the packets generated by the same host). In the context of our work, we define the flow as the maximum ”transport unit” of an attack. In other words, an attack can not span multiple flows. Thus, by requiring from our load balancing algorithm to be flow preserving, we mean that we want the packets belonging to the same flow to be sent to the same detection engine, otherwise an attack could be missed. It is therefore an essential requirement that the traffic splitting algorithms preserve the flows. In Chapter 3 we present our flow definition.

3
Architecture

If we attempt to dissect the functionality of a NIPS, we will conclude that it is assembled by two primary components: the component that is responsible for forwarding the packets from/to the network, and the component that is responsible for the analysis of the traffic. These are two components with very different characteristics. The first component is data-intensive, while the second is processor-intensive. Particularly, the first component requires from a NIPS to have a data-path with high bandwidth and low latency, while the second component requires a processor with high processing capacity.

Notice that there are two types of processors that match exactly these characteristics. On one hand we have network processors that are designed for fast packet forwarding. On the other hand we have general purpose processors that have a remarkable amount of processing power. To this effect, our NIPS, called *Digenis*, tries to combine these two types of processors to build a NIPS that will take advantage of the special capabilities of each processor.

As we have already mentioned, we want *Digenis* to be a high-performance, low-cost, flexible, and

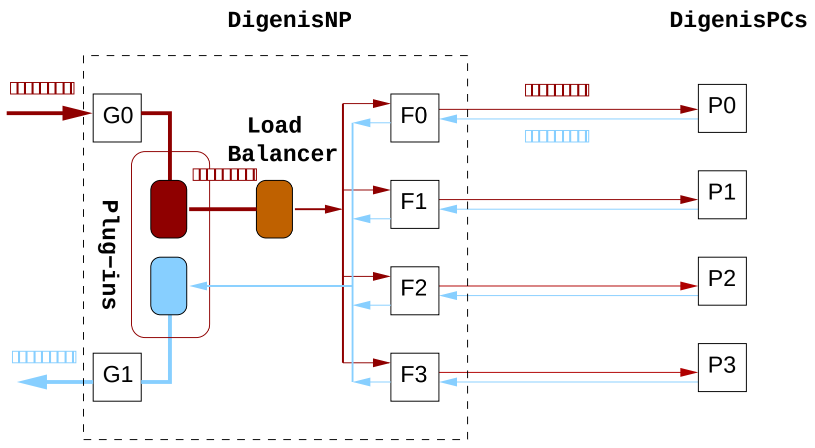


FIGURE 3.1: *Digenis* Architecture.

scalable NIPS. However, most of these goals are contradictory resulting in many tradeoffs. First, *Digenis* should exhibit adequate performance. The processing power required by a NIPS necessitates the use of multiple NPs and GPPs to support high data rates. However, we can not use as many processors as we would like because this contrasts the low-cost goal. In particular, provided that the NPs are more expensive than GPPs, this means that we have to use as few NPs as possible and try to move much of the functionality of a NIPS to the GPPs keeping the NPs simple. But if we move a portion of the functionality of a NIPS to the NP, we may be able to reduce the GPPs required considerably. Unfortunately, this is neither a flexible nor a scalable solution, especially if the portion transfered to the *IXP1200* changes frequently. These tradeoffs guided our design of *Digenis* as described next.

3.1 Architecture of *Digenis*

Digenis (Figure 3.1) is composed of an NP board (*DigenisNP*) and a number of PCs (*DigenisPCs*) connected with *DigenisNP*. *DigenisNP* is the entry and exit point of the traffic that runs through the system. The basic task of *DigenisNP* is to evenly distribute the traffic across *DigenisPCs* and to transmit the friendly packets back to their destination. *DigenisPCs* are responsible for the heavy task of inspecting the traffic for intrusion attempts. They maintain the required information for recognizing all the malicious traffic and deciding whether to forward or drop the packet. For every input packet, *DigenisNP* computes which of the *DigenisPCs* will be responsible to analyze this packet. Then it forwards the packet to a *DigenisPC* for inspection. *DigenisPC* searches for known attack patterns contained in the packet. If a pattern is found, then the packet is blocked, otherwise the packet is forwarded back to *DigenisNP*. *DigenisNP* receives the analyzed packet and transmits it to its destination.

Additionally, *Digenis* supports plug-ins that implement operations necessary to improve the performance of the system. A plug-in has two parts, one running on *DigenisNP* and one running on *DigenisPCs*. These two parts cooperate in order to accomplish their task. In the context of this work we design two plug-ins for *Digenis*. The first one attempts to minimize the cost of sending a packet from a *DigenisPC* to *DigenisNP*, while the second one tries to minimize the cost of sending a packet from *DigenisNP* to a *DigenisPC* and the cost for the *DigenisPC* to examine this packet for instrusion attempts.

3.1.1 Architecture of the NP-part of *Digenis*

The functionality of *DigenisNP* can be divided into the basic operations and the plug-ins that provide adequate operations to boost performance. The basic part of *DigenisNP* intergrates the functionality of a load balancer – it is responsible for distributing the incoming traffic across the output interfaces (ports). However, it differs from a common load balancer in that it must be flow-preserving, that is, all the packets belonging to the same flow must be forwarded to the same output interface.

3.1.2 Architecture of the PC-part of *Digenis*

A *DigenisPC* is a common PC that runs a modified popular NIDS and is connected with *DigenisNP* (through an Ethernet connection). A *DigenisPC* receives traffic from *DigenisNP* and analyzes it for possible known attacks. In case that an attack is found, it notifies *DigenisNP* to block the offending packet(s), otherwise it informs *DigenisNP* that the packet(s) should be forwarded.

A *DigenisPC* maintains state about the traffic it analyzes in order to operate correctly. The maintained state includes the active TCP connections it has captured in the near past, TCP connections tagged as offending, fragmented packets and statistics about the connections per second to destination ports.

3.2 Description of the Basic Functionality of *Digenis*

3.2.1 Definition of a Flow

There are a couple of factors we must take into account when choosing the most appropriate definition of a flow for our system. First, *DigenisNP* must be able to classify packets into flows at high speeds. Moreover, the definition must neither be too broad nor too narrow, because, on one hand we may import unnecessary load on the system and one the other hand we may miss an attack. We will try to explain what we mean with an example. Suppose that we define a flow as all the packets that originate from the same IP address. This classification can be accomplished at high-speeds, however, if these packets originate from a corporate network that uses Network Address Translation (NAT) then the load distribution might be uneven. In contrast, suppose that we define, in case of HTTP protocol, a flow to be all packets that have the same source/destination IP address, the same source/destination TCP ports, and they contain different HTTP requests (e.g, a web browser that pipelines many HTTP requests using a single TCP socket and we want to be able to recognize the different requests in order to route them to different *DigenisPCs* for better load distribution). This classification may not be accomplished at high-speeds, because in order to be able to recognize the different HTTP requests we must perform HTTP protocol decoding on *DigenisNP*.

We decided to use the following flow definition in case of TCP or UDP transport protocol. A flow is consisted of all the traffic originating from the same source IP address and TCP/UDP port and destined to the same destination IP address and UDP/TCP port. In other words, we define a flow to be a TCP or UDP connection. In case of traffic which is neither TCP nor UDP, a flow consists of all the traffic originating from the same source IP address and destined to the same destination IP address. This becomes more clear if we recall how a NIPS operates. A NIPS captures packets off the wire in order to determine what is happening on the hosts it is protecting. A packet, by itself, is not as significant to the system as the manner in which the host receiving that packet behaves after processing it. NIPSes work by predicting the behavior of networked hosts based on the packets they exchange. However, the applications running on the host and which might be vulnerable to attacks, do not communicate with other applications by directly handling packets. In contrast, they use a higher-level protocol such as TCP that is responsible for generating the packets and then assembling them back. Thus, the first thing that the NIPS does when receiving packets is to try to reassemble them and emulate the data that the destination application will receive. Thus, the NIPS predicts the befavior of the application running on the host with more accuracy and is more robust to evasion attacks [32]. Consequently, this flow definition satisfies the requirement that an attack can not span multiple flows, given that almost all the detection heuristics present in the most popular NIDSes/NIPSes, detect only attacks in the same TCP/UDP connection. However, there is one exception that we discuss later (Section 3.4).

Moreover, there is another reason that dictates this decision about the flow definition. In particular, it is highly desirable that packets belonging to the same TCP connection be processed by the same detection engine. This is due to the fact that, if the packets belonging to the same TCP connection are processed by different detection engines, then, with high probability there will be packet reordering. Packet reordering within a TCP connection may result in undesirable packet retransmissions due to the way that TCP works. Thus, if we have used another flow definition a mechanism to prevent flow reordering must have been implemented on *DigenisNP*.

3.2.2 Load Balancing Algorithm Used by *DigenisNP*

Concerning load balancing, there are two popular approaches: stateful load balancing that requires from *DigenisNP* to hold state about the mapping of flows to *DigenisPCs* and hash-based load balancing [6, 35, 22] that experiences greater load imbalances. The algorithms more suitable for high-speed flow-preserving load balancing are the hashing-based traffic splitting algorithms. These algorithms combine all the requirements mentioned in Chapter 2 and offer the best tradeoff. For the purposes of this work, we assume that load imbalances are tolerable and we use the simple hash-based method. The input of the hash function is composed of the source and destination IP addresses of the packet and the source and destination port addresses (in case of UDP or TCP). However, the TCP/UDP port information may not be available if the packet is fragmented. In such a case, *DigenisNP* waits until all the fragments are received, then reconstructs the original IP packet, reads the TCP/UDP ports, and decides the destination *DigenisPC*. Then, *DigenisNP* forwards to a *DigenisPC* the fragments of the IP packet. We did not forward to *DigenisPCs* the defragmented packets because we want our NIPS to be completely transparent. Besides, we may hide useful information from *DigenisPCs* if we forward the defragmented packets.

Of course, there is a performance issue if the rate of the IP fragments is high. Studies [39] presented that the percentage of the IP traffic that is fragmented is less than 5%. If this percentage grows above acceptable thresholds, then *DigenisNP* can simply use only the IP address information for the load balancing, which may increase load imbalance in the case of highly fragmented IP traffic [1].

[1] In other words, in this case *Digenis* changes the fbw defi nition.

Hashing Algorithm

The hashing algorithm used by *DigenisNP* uses binary polynomial multiplication and division under module-2 addition. The input of the algorithm is a 64-bit value and is considered to represent the coefficients of an order 63 polynomial in x. The input polynomial $A(x)$, is multiplied by a hash multiplier $M(x)$, using a modulo-2 addition. Since the multiplication is performed using modulo-2 addition, the result polynomial has an order of 126 with coefficients that are 1 or 0. This product is divided by a fixed generator polynomial $G(x)$, and the result is an order 64 polynomial $Q(x)$ and a remainder polynomial $R(x)$ of order 63. The above polynomials and the operations performed are shown in Equations 3.1, 3.2, 3.3, and 3.4.

$$A(x) = a_{63} * x^{63} + a_{62} * x^{62} + ... + a_2 * x^2 + a_1 * x + a_0 \tag{3.1}$$

$$M(x) = m_{63} * x^{63} + m_{62} * x^{62} + ... + m_2 * x^2 + m_1 * x + m_0 \tag{3.2}$$

$$G(x) = x^{64} + x^{63} + x^{35} + x^{17} + 1 \tag{3.3}$$

$$\frac{A(x) * M(x)}{G(x)} = R(x) + Q(x) \tag{3.4}$$

The polynomial $G(x)$ is irreducible and as a result for a fixed $M(x)$ there is a unique $R(x)$ for every input $A(x)$. The polynomial $Q(x)$ can then be discarded, since input $A(x)$ can be derived from its corresponding remainder $R(x)$. The coefficients of $R(x)$ is the hash result of the input data.

Limitations of Load Balancing

As we have mentioned, the load balancing algorithm that *Digenis* uses, is flow-preserving. To put it differently, the granularity of the load balancing in *Digenis* is the flow. Although, this granularity is fine under normal circumstances there is a possibility that an attacker could exploit this weakness of *Digenis* and create huge flows that will overload some *DigenisPCs*. In this case, the overloaded *DigenisPCs* would start loosing packets and might let an attack go undetected. While theoretically such an attack is possible, in practice it is very difficult to create these types of attacks, because in order to generate so much traffic to overload *DigenisPCs*, you need thousand or even million of compromised machines to use as traffic generators [2].

Additionally, it is important to note that the presented algortihm provides load balancing in case that the traffic received by *Digenis* is uniformly distributed over the five-tuple object space. In contrast, the loads due to the actual traffic received at *Digenis* may, by no means, be distributed uniformly over this object space, but rather will exhibit certain locality patterns. That means that despite the load-balancing property of the hash function used, the mapping between flows and *DigenisPCs* can potentially lead to imbalanced load distributions. To overcome this problem, we can use an adaptive load balancing technique such as the one presented in [21].

3.3 Description of the Plug-ins of *Digenis*

3.3.1 A Plug-In for Reducing Redundant Packet Transmission

In this Section we describe the first of the two plug-ins we designed as part of this work. This plug-in is responsible for reducing redundant packet transmission on the system. The idea behind this plug-in is the following: suppose that *DigenisNP* stores temporarily (for a few milliseconds) the packets that it forwards to *DigenisPCs* for analysis. Then there is no need for *DigenisPCs* to send back to *DigenisNP* the analyzed packet, but only a unique identifier of that packet. Because *DigenisNP* has previously stored the packet with this unique identifier, it can infer the referenced packet and forward it to the appropriate destination. The only extra work for *DigenisNP* is to tag each packet with a unique identifier, which is a trivial task. Although the additional processing cost to *DigenisNP* from this plug-in is minimal, the reduction to the load of the *DigenisPC* is surprising. However, this technique requires from *DigenisNP* to be equipped with additional memory for the buffering of the packets. As we will present in Chapter 5 the memory requirements are easily satisfied by modern NPs. Subsequently, we discuss how a *DigenisPC* communicate the packet information back to *DigenisNP*.

Communication between *DigenisNP-DigenisPC*

DigenisNP communicates with *DigenisPCs* in order to decide the action that should be performed, that is, forward or drop the packet. This is done with acknowledgments (ACKs) from *DigenisPCs* to *DigenisNP*. An ACK is an ordinary Ethernet packet. It consists of an Ethernet header, followed by two

[2]Although such an attack is diffi cult, is not impossible. Internet worms can easily perform such attacks.

bytes denoting the number of packets acknowledged (ACK factor), followed by a set of four-bytes integers representing the *PID*s (Figure 3.2). There are other possible formats requiring less bytes and supporting higher ACK factors for this configuration. However, this approach is more scalable.

Dst MAC	Src MAC	Proto	P-CACK Factor	PID 1	PID 2	...	PID N

FIGURE 3.2: ACK Packet Format.

There are several options regarding the information that these packets should contain. *DigenisPCs* may send back to *DigenisNP* the following responses:

1. **Positive ACKs:** an ACK for every packet not related to any intrusion attempt.

2. **Positive cumulative ACKs:** an ACK for a set of packets not related to any intrusion attempt.

3. **Negative ACKs:** an ACK for every packet that belongs to an offending session.

4. **Negative cumulative ACKs:** an ACK for a set of packets that belong to an attack session.

5. **The packet received.**

Each of these solutions has its pros and cons. The packet received (PR) scheme, although it has the advantage that it does not require the NP to temporary hold the packet in memory, it suffers from low performance. In Chapter 5, we evaluate some of these approaches, with regard to performance. Among positive and negative cumulative ACKs (CACKs) we have chosen the former ones. Negative CACKs have two major drawbacks: First, in order to be able to distinguish when a packet must be forwarded, we have to use a timeout value. Recall that, our NIPS must not drop any packet or an attack might be missed. As a result, we would be forced to choose a timeout for the worst case scenario. The side-effect is that packets will experience a high latency. Second, it is impossible for the NP to differentiate the case where the analyzed packet contained no intrusion from the case where the packet was dropeed due to an error condition. We chose positive CACKs (P-CACKs) because they supersede positive ACKs. Table 3.1 shows the size of the ACK packets for a number of P-CACK factors.

P-CACK Factor	Packet Size (without CRC) in Bytes
≤ 8	60
16	80
64	272
128	528
256	1040

TABLE 3.1: ACK Packet Size.

Dynamic Reconfiguration of *Digenis*

As we have already mentioned, one performance goal of *Digenis* is to keep the latency imported to packets running through the system low. In Chapter 5 we evaluate the PR and P-CACK schemes with regard to the imported latency. As we demonstrate, the latency increases as the P-CACK factor grows and the PR scheme has lower latency than the P-CACK schemes. Similarly, the performance of the system improves as the P-CACK factor increases and the PR scheme has the worst performance. Moreover, as we present in Chapter 5, the difference in latency between the schemes increases as the load of the system decreases. But when the load of the system decreases, it is possible to use a scheme that requires more processing power but has lower latency.

Thus, the rule that drives the configuration of *Digenis* during runtime is that as the load of the system increases, we move from the PR to the P-CACK scheme, or we increase the factor of the P-CACK scheme. In a similar way, as the load of the system decreases we move from the P-CACK to the PR scheme, or we decrease the factor of the P-CACK scheme. In this way, we have a rule for the dynamic reconfiguration of the system, but there are possible alternative ways regarding how the system will be reconfigured.

The decision on which ACK scheme to use can be taken either by *DigenisNP* or *DigenisPCs*. We can nominate *DigenisNP* for deciding which ACK scheme to use. However, this approach has a number of disadvantages: (1) *DigenisNP* may not have all the appropriate information for an optimal decision, (2) *DigenisPCs* will be forced to use the same ACK scheme (an approach that would differentiate *DigenisPCs* is too complicated to be implemented on *DigenisNP*) and (3) we introduce unnecessary complication on *DigenisNP*. We determine that the best way is to have *DigenisPCs* decide when they want to reconfigure its ACK scheme because this approach has a number of advantages: (1) *DigenisPCs* know better than *DigenisNP* when they are overloaded, (2) each *DigenisPC* can use a different ACK scheme and this is especially useful if *DigenisPCs* are heterogenous, and (3) complicated performance metrics can be used.

There are a couple of metrics that can be used as a load indicator. One simple metric could be the utilization of the processor of the *DigenisPC*. The *DigenisPC* could periodically check (for example using the real time clock) the load of the processor and if the load increases above or below a threshold, a reconfiguration takes place. A second metric is to monitor the size of the input packet queue and in the case that a threshold is reached a reconfiguration occurs. Another metric is to measure the latency exprerienced by the analyzed packets. If the average value crosses a threshold, then *DigenisPC* is reconfigured. For instance, the operator of the system can specify that he wants the average value of the latency introduced by the system to be 10 milliseconds. Then *DigenisPCs* could monitor the average latency experienced by packets and if this value is above the 10 milliseconds threshold, then lower the P-CACK factor. However, notice that if the system is overloaded, such a decision may result in increasing the average latency. The first metric presented is better if we want the best performance from our system, while the other two are better when we want to bound the latency of the packets crossing our system. Another solution would be the combination of the above metrics. For instance, we measure the latency, and if a threshold is reached, we check the load of the processor. If the load is high, we increment the P-CACK factor (it depends on the previous value, if it is already high we do nothing), otherwise we decrement the P-CACK factor.

3.3.2 A Plug-In for Reducing Redundant Packet Inspection

In this Section we describe the second plug-in we designed as part of this work. This plug-in is responsible for reducing redundant packet inspection in the system. The idea behind this plug-in is to take advantage of the redundancy that exists at the packet level to improve the performance of *Digenis*. Santos et al. [36] have studied the redundancy at the packet level and proposed a technique to increase effective link bandwidth by suppressing replicated data. In constrast, we use the redundancy present in the packet level to avoid inspecting again a packet that we have inspected in the near past.

For instance, suppose that we want to secure a network that consists of popular web and FTP servers.

These machines serve many different clients with the same content repeatedly. To take advantage of this locality, we propose a cache that will store the results of previous packet inspections and when the system captures a packet that has been inspected before, it will use the cached result.

Replicated Traffic

Packets vs. Higher-Level Data Units In the above example the data unit to check for replication is the packet. However, a NIPS that examines only network packets can be easily fooled [32]. For example, an attacker could split the original IP packet that contained the intrusion into many IP fragments that each of them contains a part of the instrusion. A NIPS that looks only the network packets for intrusions will miss the attack. To overcome this obstacle, most of the NIPSes today search for detection heuristics not in the network packets themselves, but in higher level data units. In particular, the popular NIDS *Snort* creates a higher level data unit (*HDU*) that containts chunks of the reconstructed TCP stream. In other words the *DU* contains the payload of many packets after the fragmented IP packets have been reassembled, the retrasmitted TCP packets have been dropped, and the TCP packets have been put in order. Moreover, the *HDU* contains information regarding the connection such as IP addresses and UDP/TCP ports. In our system, in order to provide protection against evasion attacks like the one mentioned, we also consider the caching of the *HDUs*. For the rest of this thesis we will refer to both *HDUs* and packets as data units (*DUs*).

Definition of Replicated ***DUs*** We define a *DU* to be replicated when the contents of its payload match exactly the contents of a previously observed payload and trigger the same detection heuristics as the previously observed *DU*. *Digenis* uses the IP protocol and the source/destination TCP/UDP ports to find out which detection heuristics to apply. Thus, to be more specific, we define a *DU* to be replicated when the contents of its payload match exactly the contents of a previously observed payload and the IP protocol and TCP/UDP ports match.

Finding Replicated ***DUs*** Our goal is to process a stream of *DUs* and, for each input *DU*, quickly decide whether the *DU* has been observed previously. There are a couple of ways to do this. The obvious solution is to hold the *DUs* in memory and for each *DU* received, to traverse the memory comparing the received *DU* with the stored *DUs*. However, this approach is inefficient and will result is a system that has worst performance than the system that just inspects every *DU*.

Another more efficient approach is to compute a fingerprint for each *DU*. Fingerprints are integers generated by an one-way hash function applied to a set of bytes. Good fingerprint algorithms generate well-distributed fingerprints, thus each fingerprint is compact and also unique with high probability. In other words, if two fingeprints are equal then, with high probability, the data from where these fingeprints are derived, are the same. Thus, the cacheable unit is not the *DU* anymore, but the fingeprint of the *DU*.

To deduce, our relaxed definition of replicated traffic is the following: two *DUs* are replicated when the fingerprints of their payloads are the same and the IP protocol and TCP/UDP ports match. Afterwards, we present the way we use to compute the fingerprint of a *DU*.

Fingerprint Computation The fingerprint of a *DU* is generated by applying the Secure Hash Algorithm (SHA-1)[1] on the payload of the *DU*. We chose SHA-1 because is considered to be the state-of-the-art algortihm in generating collision resistance hashes. Besides, next generation network processors, such as *IXP2850* provide a SHA-1 unit for fast computations. With a message of any length ($< 2^{64}$ bits) as input, the SHA-1 produces a 160-bit output called a *message digest*. The SHA-1 is called secure because it is computationally infeasible to find a message which corresponds to a given message digest, or to find two different messages which produce the same message digest. Moreover, the probability of two inputs to SHA-1 producing the same output is far lower than the probability of hardware bit errors. Thus, we follow the widely-accepted practice of assuming no hash collisions.

Algorithm Our algorithm for reducing redundant packet inspection runs for every *DU* of a (possibly infinite) input stream. A *DU* cache (*DUCache*) is used to store the most recent fingerprints of the *DUs*, and this is the cache against which the input *DU* is checked for redundancy. The *DUCache* is indexed by the fingerprints of the *DUs* that it holds. For every *DU* received, the algorithm first generates the fingeprint. The fingerprint is checked against the index of the cache. If it is found, then a *DU* in the *DUCache* has the same content as the input *DU* and the IP protocol and TCP/UDP ports are checked for equality. If they are equal, then we have a *DUCache* hit, otherwise we have a miss. In case of a hit, the cached result of the previous inspection of the *DU* is read, otherwise the *DUCache* is updated by inserting the newly processed *DU* (fingerprint) after it has been analyzed by the detection engine[3].

[3] We use a direct-mapped cache.

Incorporation of the Plug-in into *Digenis*

This plug-in can be applied either in *DigenisNP* or *DigenisPCs*. Both approaches have pros and cons. Aftwerwards, we present both approaches. However, we describe only the case where the *DU* is a packet. The case where the *DU* is a *HDU* is similar with the one where the *DU* is a packet.

NP-part of the Plug-in If the algorithm is applied on *DigenisNP* we have the following scenario. For every packet received, the fingerprint of the packet is generated and checked for presence in the *DUCache*. If the fingerprint is found, an action is performed based on the result of the previous inspection. If the action is to block the packet, the packet is dropped otherwise the packet is forwarded to its destination without being inspected by a *DigenisPC*. If the fingerprint is not found, the packet is forwarded to a *DigenisPC* for analysis. When, in a later point of time, *DigenisNP* finds out the result of the analysis, it caches the result. The advantage in this case is that the load of *DigenisPCs* is decreased considerably. Additionally, if the processing cost of fingerprint generation and cache lookup is small, the load of *DigenisNP* may drop due to the reduction of the packets sent for analysis. However, the disadvantage is that if we want to cache not packets but *HDUs* the implementation may be difficult.

PC-part of the Plug-in If the algorithm is applied on a *DigenisPC*, then the scenario is the following. The fingeprint of every packet is generated and is looked up in the *DUCache*. If the fingeprint is present, then an action is performed based on the result of the previous analysis. If the action is to forward the packet, the packet is forwarded to *DigenisNP* without to pass through the detection engine, otherwise the packet is dropped. If the fingeprint is absent, then the packet is analyzed by the detection engine, the result is stored in the cache and an action based on the result is taken (forward or drop the packet). The advantage in this approach is the ease of the implementation. The disadvantage is that the benefit is less than in the case that the technique is applied on *DigenisNP*. This stems from the fact that we pay the processing cost of receiving the packet although the (fingerprint of the) packet is in the cache[4].

3.4 Limitations of *Digenis*

Although *Digenis* is capable of detecting a vast number of intrusion attempts, there are some types of intrusions that might be undetected. The main reason is our definition of flows, which, do not permit attacks that span multiple flows. We assume that there are no instrusions that can span multiple TCP/UDP

[4]Our experiments showed that the reception of packets in *DigenisPC* costs about the 20-30% of the total processing cost.

connections. While this assumption holds for the majority of the detection heuristics, there are some heuristcs that try to identify attacks spanning multiple TCP/UDP connections. The majority of these heuristics, try to identify patterns of activity that deviates from a pre-defined normal activity. Thus, we can say that these heuristics detect attacks more suitable for anomaly-based instrusion prevention systems. One class of intrusions that span multiple flows are the ”portscans” that are described next.

3.4.1 Portscan Detection

Attackers perform portscans of IP addresses to find vulnerable machines to compromise. We would try to present the problem with an example. Suppose that every *DigenisPC* has a detection heuristic that states that if a specific IP address *A* initiates more than 100 connections per second to a limited number of IP addresses, then the machine with the IP address *A* is performing a portscan. Assume that we have configured *Digenis* to have four *DigenisPCs* and that a specified IP address *B* performs 100 connections per second to 100 other IP addresses that are protected by *Digenis*. Then because *DigenisNP* load balances the traffic, it will approximately hand to every *DigenisPC* the packets of 25 flows. Thus, every *DigenisPC* will conceive that there is an IP address *B* that creates 25 connections per second (< 100) and, in this way, every *DigenisPC* will suppose that everything is normal and no portscan activity takes place.

A solution to this problem is to transfer the responsibility of detecting portscans from *DigenisPCs* to *DigenisNP*. This is a viable solution because the processing cost of detecting portscans is low. That happens because the portscan detection engine checks only packet headers and no payload.

4
Implementation

4.1 Implementation of the NP-part of *Digenis* on the *IXP1200* Network Processor

We have implemented the NP-part of *Digenis* using an IXP1200 network processor. The IXP1200 chip contains six microengines with four hardware threads (contexts) each. Also, this chip has a general purpose StrongARM processor core, an FBI unit, and buses for off-chip memories (SRAM and SDRAM). The maximum addressable SRAM and SDRAM memory are 8 Mbytes and 256 Mbytes respectively. The FBI unit interfaces the IXP1200 chip with the media access control (MAC) units through the IX bus. The FBI also contains a hash unit that can take 48-bit or 64-bit data and produce a 48- or 64-bit hash index. In our evaluation board an IXF440 MAC unit (with eight Fast Ethernet interfaces) and an IXF1002 MAC unit (with two Gigabit Ethernet interfaces) are connected to the IX bus.

We have developed the application using microengine assembly language. The assignment of threads to tasks is done as follows: we assign eight threads for the receive part of the Gigabit Ethernet interface, one thread for the receive part of each of the eight Fast Ethernet interfaces, four threads for the transmit part of the eight Fast Ethernet interfaces, and four threads for the transmit part of the Gigabit Ethernet interface.

4.1.1 Implementation of the Hash-Based Load Balancing

For the implementation of the hash-based load balancing we use the hash unit of the *IXP1200*. The algorithm that the hash unit implements is described on Chapter 3. Specifically, every input packet is checked to verify that it is not an IP fragment. If it is not a fragment, the source and destination IP addresses and UDP/TCP ports are send to the hash unit. Then the last 3 bits of the result specify the output interface. If it is an IP fragment, then the packet is enqueued to the StrongARM. The StrongARM drains this queue and assembles the IP fragments into a non fragmented IP packet. After the StrongARM acquires the non fragmented IP packet, there are two possible approaches. The first approach is to perform the hashing and enqueue the packet to an output queue for transmission (this queue is drained by a microengine). The second approach is to enqueue this packet to the microengines which are then responsible to perform the hashing. The first approach has the disadvantage that the hashing must be performed in software because the StrongARM has not direct access to the hash unit. Thus, if the rate of the IP fragments received is high, the StrongARM would become the bootleneck. However, in Chapter 3 we described a solution to this problem. The second approach has the advantage that the hashing is performed more efficiently. In our prototype implementation, we use the second approach.

4.1.2 Implementation of the Plug-in for Reducing Redundant Packet Transmission

As we have mentioned in Chapter 3, this plug-in stores temporarily in *DigenisNP* the incoming packets until they are acknowledged by *DigenisPCs*. Every packet in the *IXP1200* is represented by a packet descriptor and a packet buffer. The packet buffer is a memory region in SDRAM that holds the bytes of the packet. The packet descriptor holds information regarding the size of the packet and the address of the packet buffer in SDRAM and it resides in SRAM. The data structures used for holding packet buffers and its assosiative packet descriptors are circular buffers (Figure 4.1). These circular buffers must be large enough to prevent overwriting packets before their matching P-CACK is received. Concisely, in our prototype implementation, every packet buffer has a fixed size of 2048 bytes and the associated packet descriptor has a fixed size of 8 bytes. For every ACK received, the microengines read the PIDs contained in the packet. Then, they use the PID as an index into the circular buffer of packet descriptors

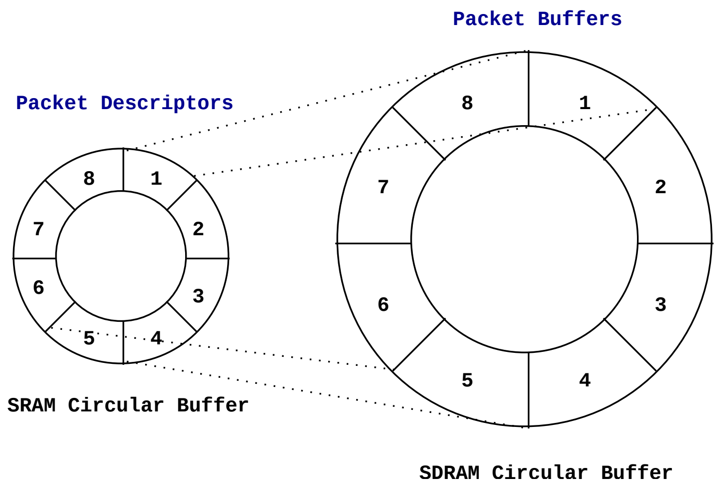


FIGURE 4.1: SRAM and SDRAM Circular Buffers for Fixed-Size Packet Buffers.

in SRAM memory to find out the SDRAM address of the packet buffer and the size of the packet.

Although we use a fixed size packet buffer of 2048 bytes leading to poor memory utilization, we could easily modify our microcode to support variable size packet buffers. In order to support this functionality, we need to know the size of the packet before requesting an unallocated packet descriptor and consequently a packet buffer. However, the *IXP1200* does not provide a way to know the actual packet size before the microengines receive even the last part of the packet. To bypass this obstacle, we can use the IP header to compute the size of the packet. Thus, every packet received from the input Gigabit Ethernet interface is stored in a packet buffer, after the size of the packet is resolved. Then, the pointer that points to the next free packet buffer is advanced by the size of the packet [1]. Although the modifications required in order to have variable size packet buffers are minor, we have not implemented them, as explained next.

Limitations

We have implemented our prototype using the Intel IXP1200 Ethernet Evaluation Board (*IXP1200EEB*). The problem is that the *IXP1200EEB* has only 1024 free packet descriptors while there are 16 Mbytes of free SDRAM[2]. Thus, the restriction to our capability to store packets is the SRAM and not the SDRAM.

[1]The SDRAM on the *IXP1200* is not byte addressable but quad-word (8 bytes) addressable, so, the pointer is advanced by the size of the packet plus some bytes for quad-word alignment.

[2]16 Mbytes are also used by the monitor running on the StrongARM.

However, in other evaluation boards such as the one provided by *Radisys* [33] there are no such restriction.

Also, in our prototype implementation we use only one circular buffer for the packet buffer allocation. This works if all the *DigenisPCs* have the same processing capabilities and the load balancing function does not create immense load imbalance. However, if *DigenisPCs* has different processing capabilities, then *DigenisPC* which have more processing power will eventualy start overwritting packet buffers in memory that belong to a less powerful *DigenisPC*. This will result in some packets to be duplicated and some packets to disappear. The obvious solution is to use one circular buffer for each *DigenisPC*. The microengines would first decide the destination *DigenisPC* and then would allocate a packet buffer to store temporarily the packet [3]. Thus, the circular buffer of a *DigenisPC* would not interfere with the circular buffer of a less powerful *DigenisPC*.

4.1.3 Implementation of the Plug-in for Reducing Redundant Packet Inspection

In our prototype system we have implemented this plug-in only for packets and not for higher level data units. Moreover, we cache the result only in the case that the packet does not contain an intrusion. So, the prototype implementation works as follows. For each input packet we first compute its fingerprint[4] by using the hash unit of the *IXP1200*. Then, we use this fingerprint as an index into the cache. If the fingerprint is contained in the cache, the packet is enqueued to the Gigabit Ethernet output port for transmission. If, in contrast, the fingerprint is not contained in the cache, the packet is sent to a *DigenisPC* for inspection. When, afterwards, *DigenisNP* receives back the ACK, it inserts the result in the cache. As you may observe, we do an insertion irrespective of the fact that the fingerprint may already be in the cache. Although someone may find it unecessary, it is much quicker to insert a fingerprint into the cache than to check first for presence and insert it in case it is absent. In this case the ACK packet must be augmented with information about the fingerprints of the packets. However, this increase on the size of the packet does not seriously affect the performance of the *IXP1200*. Moreover, if the packet of the ACK is an issue (if for example we want to support P-CACK factors greater than 128), we can force ACK packets to contain only PIDs and use a table in SRAM memory that will map PIDs to fingerprints.

Regarding the placement of the cache, we can use both the SRAM and SDRAM memories. If the

[3]It is possible for the *IXP1200* to read the packet data although the packet is not in SDRAM memory.

[4]The fi ngerprint is 64 bits long.

cache is small enough to fit into SRAM then the SRAM is used, otherwise the SDRAM memory is used. SRAM has less access latency than SDRAM and thus, a cache placed on SRAM may improve the overall performance of *Digenis*.

4.1.4 Pseudo-code Description of *DigenisNP*

The *IXP1200* receives data from a MAC through the IX bus. Since the IX bus is capable of transfering fixed size units of 64 bytes long, the MAC breaks each packet received into 64 byte quantities, called *mpackets*. Each mpacket is flagged as being the first (SOP) or the last (EOP) mpacket of the packet. If none of these flags are set, this is a middle mpacket (MOP), and if both flags are set then this is the only mpacket of the packet. Similarly, during transmission, the MAC device must compose of the whole packet from the mpackets that it receives from the IX bus.

During the reception, after the packet has been received from the IX bus, the *IXP1200* stores it in a slot of the receive FIFO (RFIFO)[5]. Since the number of RFIFO slots is only sixteen, the allocation of slots is done at design time and is the responsibility of the code running on the microengines to drain these slots at a rate that keeps pace with the line speed of each port. Similarly, during transmission, the *IXP1200* mangles the packets into mpackets and stores the resulting mpackets in a slot on the transmit FIFO (TFIFO)[6]. Likewise, it is the responsibility of the code running on the microengines to fill the transmit slot at a rate that keeps pace with the line speed of each port.

After a packet has been stored on the RFIFO, it is transfered into SDRAM and a descriptor of the packet is placed into a queue in SRAM. This queue is served asynchronously from a different micro-engine. Using different microengines for enqueuing and dequeuing prevents microengines from being idle during the time a packet is queued [40]. After dequeuing, the packet is broken down into mpackets and stored in TFIFO slots for transmission. Figure 4.2 shows the forwarding pipeline from the Gigabit Ethernet input port to the Fast Ethernet output ports, while Figure 4.3 shows the reverse path followed during P-CACK reception. Subsequently, we present a pseudocode description of the work done by the microengines. However, in order not to confuse the reader that has not previous experiences with the *IXP1200*, we do not include the pseudo-code for the packet caching plug-in.

[5]In reality RFIFO is not a FIFO but a buffer. Intel has corrected this flaw and in the IXP2400 network processor hardware reference manual is called RBUF.

[6]In reality TFIFO is not a FIFO but a buffer. Intel has corrected this flaw and in the IXP2400 network processor hardware reference manual is called TBUF.

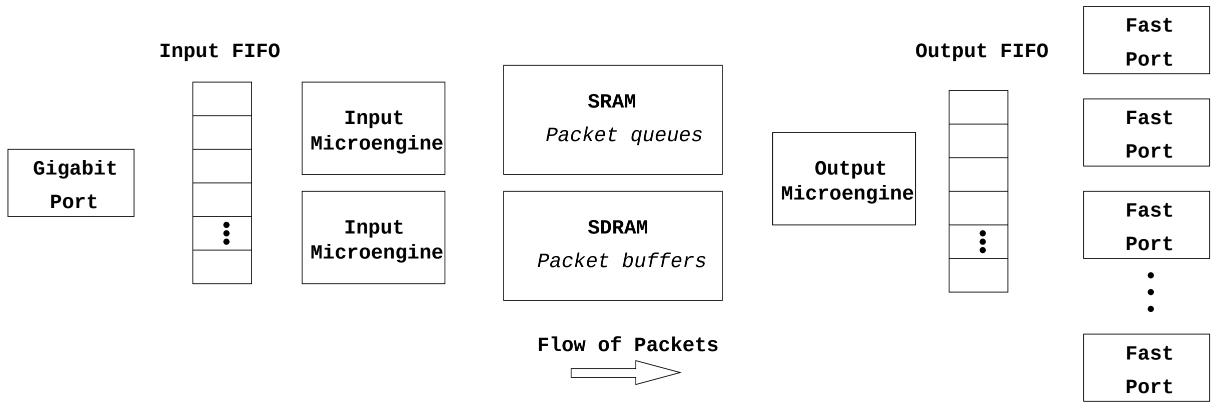


FIGURE 4.2: The Forwarding Pipeline from the Gigabit Ethernet Port to the Fast Ethernet Ports.

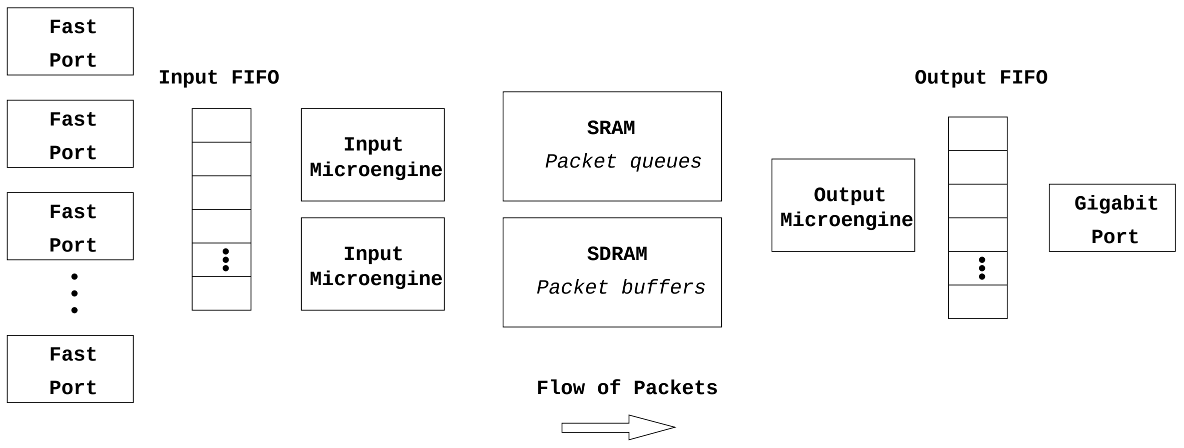


FIGURE 4.3: The Forwarding Pipeline from the Fast Ethernet Ports to the Gigabit Ethernet Port.

Gigabit Ethernet Input Processing Figure 4.4 shows our pseudo-code for the loop executed once for each mpacket received: p denotes the port number on which the mpacket arrived, c is an index in the receive FIFO, *mp_addr* is the address in memory where the contents of the mpacket are stored, *reg_mp_data* denotes the microengine registers that hold the mpacket, *state* is a data structure containing information about how the mpacket should be processed, and *out_port* is the Fast Ethernet port to which the packet will be sent.

The first set of operations (lines 1-6) determine whether port p has a new mpacket available. If so, the load operation instructs the receive state machine (RSM) to copy the mpacket from the off-chip port memory into the on-chip receive FIFO. Also, the characteristics of the mpacket are read from the RSM (SOP, MOP, EOP, number of bytes etc) and stored into *state* variable. There is only one

```
INPUT_LOOP:
crit_sect_enter();
send_rcv_request(p);
if(get_rcv_status() != PORT_HAS_DATA){
        crit_sect_exit();
        goto INPUT_LOOP;
}
load RFIFO[c];
state = get_state();
crit_sect_exit();
mp_addr = calc_mp_addr();
if(at_start_of_packet(state)){
        reg_mp_data = RFIFO[c];
        out_port = load_balance(reg_mp_data);
        SDRAM[mp_addr] = reg_mp_data;
}
else{
        SDRAM[mp_addr] = RFIFO[c];
}
if(at_end_of_packet(state)){
        enqueue(p,out_port);
}
goto INPUT_LOOP;
```

FIGURE 4.4: Pseudo-code Running in Each Context Assigned to Gigabit Ethernet Input Processing.

receive state machine on the *IXP1200* and requests concerning it are not hardware-serialized. Thus, the critical Section operations are needed to allow multiple microengine contexts to safely execute input loops in parallel. In case this is the first mpacket, the microengine copies a portion of the mpacket into its registers for further processing. Then, the load balancing code is called which specifies the port on which the packet will be forwarded. Additionally, the mpacket (with possible modifications) is transfered to the SDRAM. If the packet is not a SOP mpacket, it is transfered directly from the RFIFO to the SDRAM without the intervention of the microengine. The last step is to enqueue the packet and some identification information in the appropriate output queue in case of an EOP mpacket.

Fast Ethernet Input Processing Figure 4.5 gives pseudo-code for the loop executed once for each mpacket received. In the figure, *p_addr* is the address in memory where the contents of the packet is

```
        INPUT_LOOP:
1       crit_sect_enter();
2       if(port_rdy(p) != PORT_HAS_DATA){
3               crit_sect_exit();
4               goto INPUT_LOOP;
5       }
6       load RFIFO[c];
7       state = get_state();
8       crit_sect_exit();
9       mp_addr = calc_mp_addr();
10      SDRAM[mp_addr] = RFIFO[c];
11      if(at_end_of_packet(state)){
12              p_addr = calc_p_addr();
13              reg_p_data = SDRAM[p_addr];
14              process_ack(reg_p_data);
15      }
16      goto INPUT_LOOP;
```

FIGURE 4.5: Pseudo-code Running in Each Context Assigned to Fast Ethernet Input Processing.

stored and *reg_p_data* denotes the microengine registers that hold the packet.

The first set of operations (lines 1-5) determine whether port *p* has a new mpacket available. If so, the load operation instructs the RSM to copy the mpacket from the off-chip port memory into the on-chip RFIFO. Once the mpacket is in the RFIFO, the microengine instructs SDRAM to transfer the mpacket into SDRAM and waits for the completion signal. After receiving the completion signal, if the mpacket is the EOP mpacket, the whole packet is read to microengine registers and parsed. For every number read, the stored packet with this number as identidier is sent to the Gigabit Ethernet port.

Gigabit Ethernet Output Processing *Digenis* uses one microengine to transmit packets to the Gigabit Ethernet port. The microengine uses one thread as a scheduler to determine if there are packets on the transmit queue, and three threads, referred as fill threads, to transmit the packet data. The fill threads transfer data from SDRAM to the TFIFO and write TFIFO control data for each mpacket.

Figure 4.6 gives pseudo-code for the scheduler thread. In the figure, *mp_counter_addr* is the address in memory where the number of mpackets received is stored, *mp_counter* is the value stored in that address, *assigned_mp_counter* is a counter shared by all threads in the microengine that counts the number of

```
void tx_gig_assign(sem_tid, assign){
        if(mp_counter == assigned_mp_counter){
                tmp = SKIP;
        }
        else{
                assigned_mp_counter = assigned_mp_counter + 1;
                tmp = PROCESS;
        }
        semaphore_wait(sem_tid);
        assign = tmp;
}

SCHEDULER_LOOP:

mp_counter = SCRATCH[mp_counter_addr];
tx_gig_assign(sem1, assign1);
tx_gig_assign(sem2, assign2);
tx_gig_assign(sem3, assign3);

goto SCHEDULER_LOOP;
```

FIGURE 4.6: Pseudo-code of the Scheduler Thread for Gigabit Ethernet Output Processing.

mpackets processed by the scheduler thread, *assign[1-3]* are registers used for communication between the scheduler thread and the fill threads where information about the packet to be transmitted are stored, and *sem[1-3]* are semaphores used to protect access to *assign[1-3]* registers.

The transmit queue of the Gigabit Ethernet port has associated with it a counter that resides in scratch-pad memory. When this counter (*mp_counter*) is greater than the *assigned_mp_counter*, this implies that there is a new packet on the queue, or a packet with mpackets still to be transmitted. The scheduler sends an assignment to a fill thread, which transmits at most one mpacket at a time. The scheduler continues to assign fill threads to the packet until all of the mpacket that make up the packet are transmitted to the outbound Gigabit Ethernet port. If the two counters are equal, then the scheduler determines that there are no mpackets that require transmission and issues an assignment to the fill thread with a skip command. Without deeping into too much details, the skip command ensures that the transmit state machine advances to the next TFIFO element which may contain data from another transmit microengine, for example the microengine that transmits packets to the eight Fast Ethernet ports.

```
void wait_for_assign(assign){
       WAIT_LOOP:
       if(assign != SKIP && assign != PROCESS)
              goto WAIT_LOOP;
}

FILL_THREAD_LOOP:
wait_for_assign(assign1);
skip = check_skip(assign1);
sem_wakeup(sem1);
if(skip == TRUE)
       process_skip();
else
       process_assign();
goto FILL_THREAD_LOOP;
```

FIGURE 4.7: Pseudo-code of the First Fill Thread for Gigabit Ethernet Output Processing.

Figure 4.7 gives pseudo-code for the first fill thread. In the figure, *skip* denotes whether the assignment is a skip assignment or not. The fill threads read assignments from the *assign[1-3]* registers and determines whether the assignment is to transmit an mpacket or to process a skip assignment. If, the fill thread has an mpacket to process, the *process_assign* function is called that transfers the mpacket from the SDRAM to the TFIFO. Otherwise, the *process_skip* function is called that advances to the next TFIFO element.

Fast Ethernet Output Processing The transmit code for the Fast Ethernet ports is similar with that of the Gigabit Ethernet port. The only difference is that the counter that keeps track of the number of mpackets received is replaced with a bit vector. If the bit is set, then the transmit queue (and consequently port) that this bit represents has available packets for transmission, otherwise it has not queued packets.

4.2 Implementation of the PC-part of *Digenis*

The functionality of *DigenisPCs* is implemented by modifying the popular NIDS *Snort* [34]. We modify *Snort* so that in case that the inspected packet is friendly, the packet is send back to the *DigenisNP*[7]. The implementation of the two plugins are described next.

[7]For the transmission of packets from the PC to the NP we use *libnet* [37].

4.2.1 Implementation of the Plug-in for Reducing Redundant Packet Transmission

For this plug-in to work, we modify *Snort* to send a P-CACK packet back to the *DigenisNP* if no attack is identified. *Snort* accumulates the PIDs of the friendly packets into a buffer (which has the format of an Ethernet packet) and when the specified number of packets are inspected and do not relate to an attack, the buffer is forwarded to *DigenisNP*.

4.2.2 Implementation of the Plug-in for Reducing Redundant Packet Inspection

This plug-in has been implemented as a *Snort* preprocessor. In contrast to the *DigenisNP* version of the plug-in, this plugin is able to cache both packets and *HDUs*. Moreover, someone can choose to cache only traffic originated from or destined to servers. Given that the traffic originating from servers has more redundancy than the traffic originating from clients, this option can be used to improve performance.

4.3 Implementation of Additional Confi gurations

In addition to *Digenis* configuration, for comparison pusposes, we have implemented the following three configurations. A forwarder (FWD) that transmits the traffic arriving at an input Gigabit Ethernet interface to an output Gigabit Ethernet interface. A load balancer (LB) that implements a flow-preserving load balancer with the same load-balancing characteristics as *Digenis*. The *IXP1200* receives traffic from a Gigabit Ethernet interface and transmits the traffic to eight Fast Ethernet interfaces. The last configuration (LB + FWD) implements the basic functionality of *Digenis* (without optimizations).

5 Evaluation

In this Chapter we examine the performance of our architecture. First, we examine the impact of our enhancements to *DigenisPC-DigenisNP* communication by offloading *DigenisPCs* with reducing redundant packet transmission. Second, we examine the performance improvement we achieve by reducing redundant packet inspection.

5.1 Evaluation of the Plug-in for Reducing Redundant Packet Transmission

In this Section we focus on *DigenisPC-DigenisNP* coordination. In particular, we compare the performance of P-CACK vs. the PR scheme. We also show that such techniques can be efficiently supported by current NPs [1] and that they do not significantly impair forwarding latency.

[1]In fact, the *IXP1200* should be considered as a cheap, low-end device.

5.1.1 Experimental Environment

Experimental Environment for *DigenisNP*

The performance of the configurations running on the *IXP1200* is measured using the Developer Workbench (version 2.01a). Specifically, we use the *transactor* provided by Intel. The transactor is a cycle-accurate architectural model of the *IXP1200* hardware. We simulate the configurations as they would run on a real *IXP1200* chip. We assume a clock frequency of 232 MHz and a 64-bit IX bus with a clock frequency of 104 MHz.

Experimental Environment for *DigenisPCs*

We use a 2.66 GHz Pentium IV Xeon processor with hyper-threading disabled. The PC has 512 Mbytes of SDRAM memory at 133 MHz. The PCI bus is 64-bit wide clocked at 66 MHz. The host operating system is Linux (kernel version 2.4.22, Red-Hat 9.0). The Gigabit Ethernet card is an Intel PRO/1000 MT Dual Port Server Adapter [14]. We increase the buffers allocated by the driver to the maximum to achieve the best possible performance.

The software running on the PCs is a modified *Snort* version 2.0.2, compiled with gcc version 3.2.2. We turn off all preprocessing in *Snort*. Unless noted otherwise, *Snort* is configured with the default rule-set.

Packet Traces

For the evaluation of *Digenis* we use three packet traces. The **FORTH.WEB** trace was captured at **ICS-FORTH** and contains only HTTP traffic. The **FORTH.LAN** trace was also captured at **ICS-FORTH** and contains traffic from an internal Local Area Network (LAN). Both traces contain the real payload of the packets[2]. The **IDEVAL** traces are taken from MIT Lincoln Laboratory and were used in 1998 DARPA Intrusion Detection Evaluation [25].

Limitations

Before presenting the results from the evaluation of our system, we want to deal with some limitations of our experimental methodolgy. The first issue we want to discuss is the assumption made. That traffic characteristics present in our links ressembles traffic characteristics of faster links. In particular, traces

[2] We used *tcpdump* to capture the traces.

Trace	Total Packets	Total MBytes	Average Packet Size (bytes)
FORTH.WEB	2678445	2104	785
FORTH.LAN	1597396	974	610
IDEVAL	1492331	317	212

TABLE 5.1: Characteristics of the Packet Traces.

Packet Size	Percentage (%)
$size \leq 64$	40
$64 < size \leq 128$	3
$128 < size \leq 256$	1
$256 < size \leq 512$	2
$512 < size \leq 1024$	3
$1024 < size \leq 1518$	51

TABLE 5.2: Packet Size Distribution for **FORTH.WEB** Trace.

Packet Size	Percentage (%)
$size \leq 64$	25
$64 < size \leq 128$	14
$128 < size \leq 256$	20
$256 < size \leq 512$	1
$512 < size \leq 1024$	1
$1024 < size \leq 1518$	37

TABLE 5.3: Packet Size Distribution for **FORTH.LAN** Trace.

Packet Size	Percentage (%)
$size \leq 64$	69
$64 < size \leq 128$	12
$128 < size \leq 256$	5
$256 < size \leq 512$	3
$512 < size \leq 1024$	1
$1024 < size \leq 1518$	9

TABLE 5.4: Packet Size Distribution for **IDEVAL** Trace.

captured at **ICS-FORTH** are from a 10 Mbit/s LAN. However, during the evaluation we generate the same traffic as these traces but in rates that are up to 30 times higher. However, while we emulate certain conditions present in faster links, such as the content of the packet, we also miss critical information such as the order of packets and packet inter-arrival times. In particular, if the same traffic captured at a 10 Mbps link were to be transfered across a Gigabit link, then the packets would be multiplexed with packets from other network sources. Suppose that you have the traces captured at ten 100 Mbps Ethernet links and the trace from a Gigabit Ethernet link that was transfering exactly the same information (e.g with port mirroring) as the ten 100 Mbps links. Suppose also, that you concatenate the ten traces into a bigger one and you run *Snort* on top of these two traces (the concatenated one and the one captured at the Gigabit Ethernet link). Would the maximum loss free rate of *Snort* be the same in each trace? The answer is that the rates would be different.

To understand this, assume that half of the 100 Mbit links were transfering 1518-byte packets and half of them 64-byte packets. While the concatenated trace would contain chains of 1518-byte and 64-byte packets the Gigabit trace would contain 64-byte and 1518-byte packets interleaved. Remember that the 1518-byte packets require much more processing power to be analyzed than 64-byte packets because *Snort* analyzes not only the headers of the packets but also the payload. Thus, while in the first case *Snort* would have periods with peaks in loads and periods that is almost idle, in the second case it would experience an almost uniform distribution of the load over time.

Notice that systems like IP routers and firewalls are not so sensitive because they require almost the same amount of work for each packet irrespective of its size [3]. The lesson is that evaluation of a NIPS is not a straightforward process and that a number of parameters must be considered to avoid inaccuracies.

5.1.2 Performance of *DigenisNP*

In this Section we try to look into all the possible bottlenecks of the *DigenisNP*. For this reason we examine extensively the utilization of all the major components of the *IXP1200*. Specifically, we check the microengines, the SRAM and SDRAM memories, the FBI unit and the hash unit.

Utilization of the Microengines, SRAM and SDRAM

Consider that all the configurations described in Chapter 4 handle at most the IP and UDP/TCP header of the incoming packets. We argue that the most demanding traffic for the *IXP1200* is the traffic consisting of a big fraction of small packets, namely 64-byte packets [4]. We simulate the above configurations and the results signified that all the configurations were capable of sustaining line speed even with traffic

[3]They analyze only headers of the packets, not payload.

[4]This is the smallest possible packet in an Ethernet link including the 4-byte Ethernet CRC.

Configuration	ME0	ME1	ME2	ME3	ME4	ME5	SDRAM	SRAM
FWD	41.5	41.2	-	-	-	67	16	20.6
LB	52.6	52	-	-	72.2	-	18.3	24.5
Digenis (P-CACK 8)	52.2	50.9	66.9	66.8	71.5	67.7	28.4	33.5
LB+FWD	51.9	50.7	57.7	57.7	71.4	70	35.1	34.7

TABLE 5.5: Utilization(%) of the Microengines, SDRAM and SRAM for 64-byte Packets.

Configuration	ME0	ME1	ME2	ME3	ME4	ME5	SDRAM	SRAM
FWD	35.4	35.2	-	-	-	64.8	20.5	9.2
LB	38.6	38.3	-	-	73.8	-	21.3	10.9
Digenis (P-CACK 8)	37.9	37.3	68.1	68	71.8	64.8	34.5	18.3
LB+FWD	37.4	37.5	57	57	72.4	65.4	43.4	12.4

TABLE 5.6: Utilization(%) of the Microengines, SDRAM and SRAM for 512-byte Packets.

Configuration	ME0	ME1	ME2	ME3	ME4	ME5	SDRAM	SRAM
FWD	34.5	34.5	-	-	-	64.5	21	8.4
LB	36.4	36.3	-	-	73.8	-	21.6	9.4
Digenis (P-CACK 8)	35.5	35.5	68.2	68.2	72	64.6	35.1	16.6
LB+FWD	36.3	36	56.3	56.9	72.1	64.8	44.2	12.5

TABLE 5.7: Utilization(%) of the Microengines, SDRAM and SRAM for 1024-byte Packets.

Configuration	ME0	ME1	ME2	ME3	ME4	ME5	SDRAM	SRAM
FWD	34.3	34.3	-	-	-	64.6	21.1	8.2
LB	35.8	35.8	-	-	73.8	-	21.8	9
Digenis (P-CACK 8)	35.1	35	68.2	68.2	72.2	64.6	35.4	16
LB+FWD	35.4	35.3	57.6	57.6	73.0	67.2	44.6	9.2

TABLE 5.8: Utilization(%) of the Microengines, SDRAM and SRAM for 1518-byte Packets.

Configuration	ME0	ME1	ME2	ME3	ME4	ME5	SDRAM	SRAM
FWD	35.3	35.2	-	-	-	65.2	20.6	9.2
LB	39.4	40.4	-	-	74.3	-	19.7	17.5
Digenis (P-CACK 8)	37.8	37.4	66.8	66.7	72.5	65.5	29.9	12.7
LB+FWD	40.2	39	53.9	54.0	74.2	67.1	30.3	22

TABLE 5.9: Utilization(%) of the Microengines, SDRAM and SRAM for Variable Size Packets.

consisting of only 64-byte packets [5]. This is expected as the theoretical forwarding capacity of the *IXP1200* chip is greater than 1600 Mbit/s.

Whereas the configurations sustain line speeds, we use as a metric for comparison the utilization of the microengines and the utilization of SRAM and SDRAM memories[6]. These are some of the resources that may become the bottleneck, considering that the *IXP1200* specification states that the maximum IX bus throughput is 6 Gbit/s. In Tables 5.5, 5.6, 5.7 and 5.8 we present the utilization of the microengines[7] and the utilization of the SRAM and SDRAM memories for our configurations. We observe that our approach is efficient and does not consume all the resources of the *IXP1200*, leaving headroom for even more offloading of *DigenisPCs*. Particularly, our results suggest that the extra cost of *Digenis* compared to the load balancer is affordable.

To further understand these results, we have to know what task each microengine executes. The first two microengines execute the microcode responsible for receiving traffic from the first Gigabit Ethernet interface and enqueuing the packets to the appropriate queue. This queue depends on the configuration used. In case of the FWD configuration, the packets are enqueued in the queue that holds the packets to be transmitted on the second Gigabit Ethernet interface. In the other configurations, the queue depends on the result of the load balancing algorithm. Microengines 2 and 3, in case of *Digenis* are responsible for receiving the P-CACKs, parsing them and scheduling the right packet for transmission on the second Gigabit Ethernet interface. The same microengines, in case of the LB+FWD configuration, are responsible for enqueuing the packets received from the Fast Ethernet interfaces to the queue that

[5]*Digenis* used P-CACK scheme with a factor of eight.

[6]In reality, we measure the utilization of the buses of SRAM and SDRAM memories.

[7] The microengines that have not a value were not used in the specifi ed confi guration.

holds the packets to be transmitted on the second Gigabit Ethernet interface. Microengine 4 is used for transmitting packets on the eight Fast Ethernet interfaces, while microengine 5 transmits packets towards the Gigabit Ethernet interface.

With the mapping of the tasks to microengines kept in mind, we can safely explain the above results. As we observe, as the packet size grows, the utilization of microengines 0 and 1 reduces. This results from the fact that as the packet size grows, the rate of the received packets slows down. Considering that these microengines handle only the IP header of the packets, fewer packets per time unit mean less work to do. Similarly, this observation holds and for the microengines 2 and 3 for all configurations except *Digenis*. In case of *Digenis*, the utilization of microengines 2 and 3 is almost the same[8]. This is due to the fact that the rate of the incoming P-CACK packets is the same, regardless of the size of the received packet. Moreover, we observe that the utilization of microengines 4 and 5 is practically the same no matter what the size of the packet is. This is an artifact of the way that the microcode on these microengines operate. Generally, these two microengines use polling to test whether there is data to send to the output ports and in case that no data exists, they just insert a skip value in the TFIFO of the *IXP1200*. As a result, these microengines do approximately the same amount of work irrespective of the size of the packet. Finally, we have to mention that the increased utilization of microengines 2 and 3 in the case of *Digenis* is caused by the instrumentation code we add to measure the performance of *DigenisNP*. While in the other configurations we do not add code for evaluation purposes, we are obliged to do so in the case of *Digenis*. Specifically, we add a polling loop to control the rate that the *IXP1200* received P-CACK packets because there is no way to instruct the simulator to do this with high accuracy. Additionaly, this code results in an increased SRAM utilization because it uses the SRAM bus to read a value from memory.

Moreover, someone can naively believe that the most demanding traffic for the SRAM and SDRAM memories of the *IXP1200* is the traffic consisting of a big fraction of large packets, namely 1518-byte packets. This is not true for SRAM because microengines make a transaction to SRAM only once for each packet received and this transaction is irrespective of the packet size. However, for SDRAM, the simulation with traffic consisting of only 1518-byte packets reveals that the load of SDRAM memory is about 30% higher compared to the 64-byte traffic. The reason is that while 64-byte packets are much

[8]Small variations exist because microengines are not completely independent. They share the same buses and as a consequence changes on the utilization of one microengine is possible to affect the utilization of another one.

smaller compared to the 1518-byte packets, transactions on the SRAM and SDRAM buses due to 64-byte traffic are increased compared to 1518-byte traffic. In particular, the rate that the *IXP1200* receives 64-byte traffic and 1518-byte traffic is 1190 Kpps and 65 Kpps respectively. Consequently, while the 1518-byte packets are approximately 23 times larger than the 64-byte packets, they cause approximately 18 times less bus transactions.

Furthermore, we simulate the above configurations with real network traffic. We instruct the IX bus simulator of the Developer Workbench to load the packets from a file that contains the traffic of the **FORTH.LAN** trace and measure the utilization of the microengines and memories (Table 5.9).

Configuration	ME0	ME1	ME2	ME3	ME4	ME5	SDRAM	SRAM
Digenis (P-CACK 1)	52.6	50.7	67.1	67.1	69.9	65.7	36	35.6
Digenis (P-CACK 2)	52.9	50.6	67.3	67.3	70.3	66.3	30.7	34.4
Digenis (P-CACK 4)	52.4	51.2	67	67	70.4	66.3	28.7	33.7
Digenis (P-CACK 8)	52.2	50.9	66.9	66.8	71.5	67.7	28.4	33.5
Digenis (P-CACK 16)	51.2	51.7	66.6	66.2	70.6	66.2	28.2	33.7
Digenis (P-CACK 32)	53.5	49.3	65.6	66.3	70.5	66.1	28.9	33.4
Digenis (P-CACK 64)	53	49.5	64.8	64.4	70	65.8	28.7	33.1

TABLE 5.10: Utilization(%) of the Microengines, SDRAM and SRAM for 64-byte Packets.

Additionally, we measure the utilization of the microengines and the memories of the *IXP1200* for a varying number of P-CACK factors ranging from 1 to 64. The results are shown in Table 5.10. As we observe, as the P-CACK factor increases, the microengines and memories utilization stays almost the same or slightly decreases.

Utilization of the FBI Unit

Another possible bottleneck of the *IXP1200* in general and *DigenisNP* specifically is the FBI unit. The FBI unit consists of the IX bus, the hash unit, and the scratchpad memory. The FBI unit has two queues named push and pull where the microengines can enqueue commands for execution and dequeue the results. We use as an indication of the utilization of the FBI unit the size of this queue (max queue size is 8 entries). Tables 5.11 and 5.12 show the number of entries of the queues for *Digenis* and 64-byte

Queue Entries	Percent	Cumulative Percent
0	47.8	47.8
1	21.6	69.5
2	13.4	82.9
3	8.5	91.4
4	4.9	96.3
5	2.7	99
6	0.9	99.9
7	0.1	100

TABLE 5.11: FBI Pull Queue Fullness Statistics.

Queue Entries	Percent	Cumulative Percent
0	68.3	68.3
1	18.4	86.6
2	6.8	93.4
3	3.2	96.6
4	1.3	98
5	1.1	99
6	0.8	99.8
7	0.2	100

TABLE 5.12: FBI Push Queue Fullness Statistics.

packets (P-CACK factor is 8). The results show that 91.4% of the time the size of the pull queue is less than 50%, while 96.6% of the time the size of the push queue is less than 50% (compared to the max queue sizes).

Utilization of the Hash Unit

Although, the hash unit is contained inside the FBI unit, we examine it seperately due to its significance to the *DigenisNP* performance. To check the utilization of the hash unit we perform the following exper-

Queue Entries	Percent	Cumulative Percent
0	98.5	98.5
1	1.5	100

TABLE 5.13: Hash Queue Fullness Statistics.

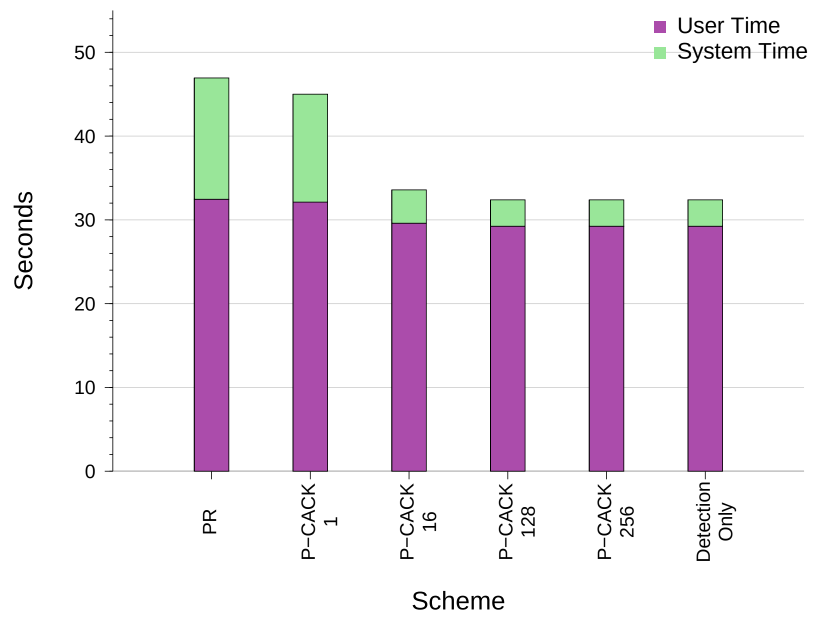


FIGURE 5.1: Processing Cost of the *DigenisPC* (**FORTH.WEB** Trace).

iment. The hash unit of the *IXP1200* has a 8-entry queue where microengines can put work assignments. The hash unit drains this queue and notifies the microengines when the assigned work is completed. We use as an indication of the utilization of the hash unit the size of the queue. Table 5.13 shows the number of entries of the queue for *Digenis* and 64-byte packets (the P-CACK factor does not affect the result). The results show that 98.5% of the time the queue of the hash unit has no pending requests while only 1.5% of the time the hash unit has one pending request.

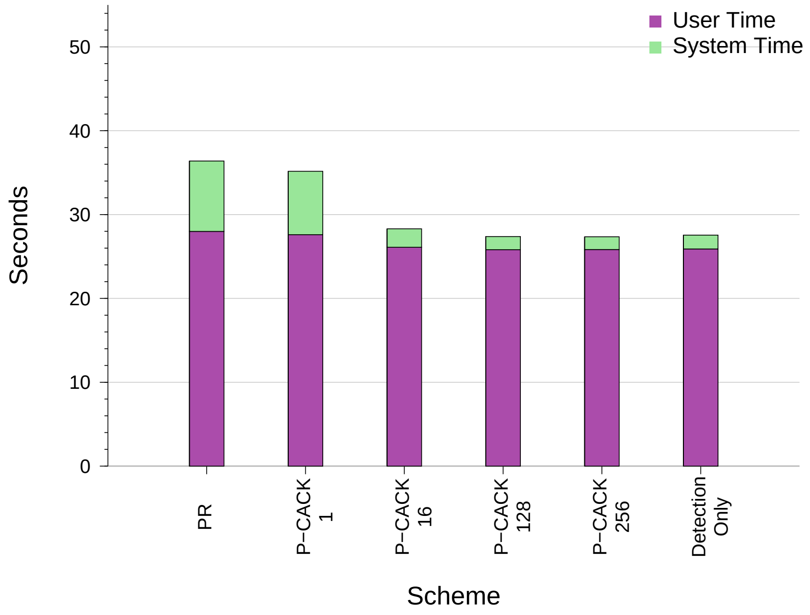


FIGURE 5.2: Processing Cost of the *DigenisPC* (**FORTH.LAN** Trace).

5.1.3 Performance of *DigenisPCs*

We first measure the processing cost of a *DigenisPC* for different coordination schemes using the default rule-set. In this experiment *Snort* simply reads traffic from a packet trace [9], performs all the necessary NIPS functionality, and then transmits the coordination messages to a hypothetical *DigenisNP* through a Gigabit Ethernet interface. Figures 5.1, 5.2 and 5.3 show the time that *Snort* spents to process all the packets for three different traces including user and system time breakdown. The results show that the bigger the P-CACK factor, the less the total running time for *Snort*. The running time is practically the same with the unmodified *Snort* for P-CACK factor equal to 128. Also, for the **FORTH.WEB** trace *Snort* finished 45% faster for P-CACK factor equal to 128 compared to PR scheme. Specifically,we observe that for the first two traces, the system time is lower than the user time. This confirms the fact that *Snort* spents most of its processing time in header and content matching which is counted in user time. This is not true in the **IDEVAL** trace because the majority of the packets of this trace are 64-byte packets. As a consequence, these packets do not require content matching, which is the single most expensive operation of a NIPS and all the user time is spent on header matching and the copying of

[9] We confi rm that the hard disk is not the bottleneck by measuring the throughput of the hard disk and the transmit rate of *Snort*. As expected, the transmit rate of *Snort* is smaller than the throughput of the disk.

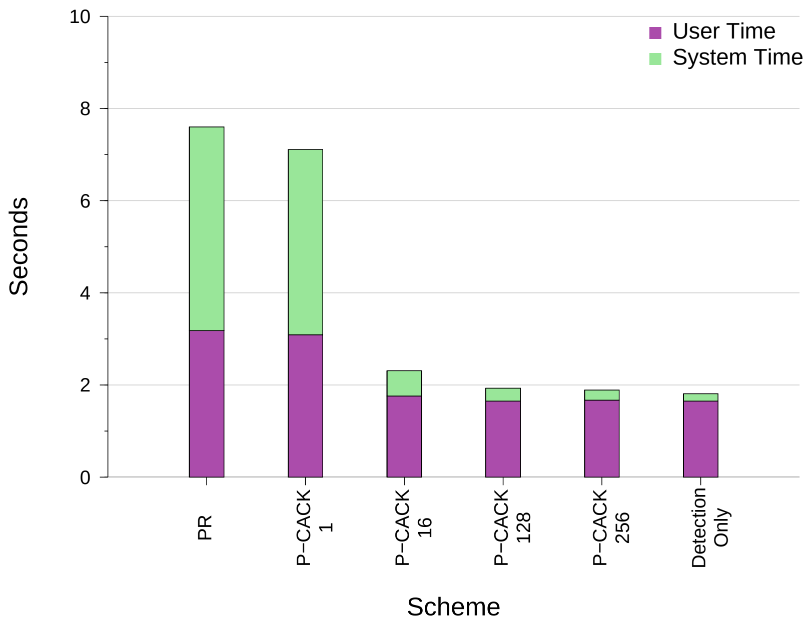


FIGURE 5.3: Processing Cost of the *DigenisPC* (**IDEVAL** Trace).

packets from user space to kernel space and vice versa. In this way, the processing cost of sending 64-byte is greater than the cost of header matching and packet copying for this trace. Additionally, notice that in the case of **IDEVAL**, the user time for the schemes PR and P-CACK with factor 1 is greater than in schemes with P-CACK factor bigger than 1. This may seem wrong but it is due to the copying of the packet from user space to kernel space memory. This phenomenon exists also in the other two traces, but due to the fact that the cost of copying is minor compared to the user time spent on header and content matching, it is more difficult to notice.

We also observe that the improvement of the P-CACK scheme compared to the PR scheme depends very much on the trace used: the P-CACK scheme was from 45% to 3 times more efficient than the PR scheme. The reason is that the improvement depends on the detection load of the *DigenisPC*. The smaller the detection load, the bigger the relative improvement. This becomes more clear if we determine where the improvement is coming from. The improvement is that it reduces the overhead required for sending a packet to network (*system time* in Figure 5.1). If the detection engine of a *DigenisPC* is overloaded, then this overhead is a small fraction of the total workload of *DigenisPC* and reducing it does not lead to much improvement. In contrast, if the the detection engine of a *DigenisPC* is lightly loaded, this overhead consumes a big fraction of the total workload of the *DigenisPC* and reducing it results in a

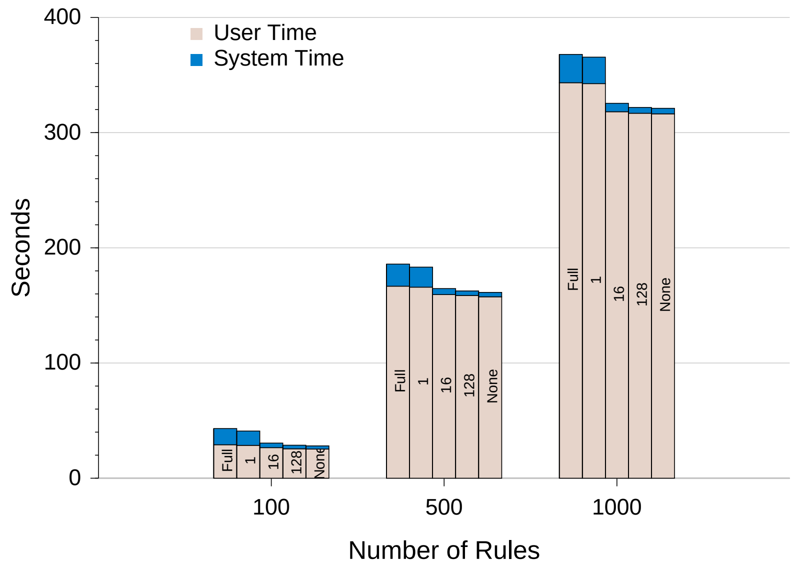


FIGURE 5.4: Performance of a *DigenisPC* using Incremental Number of Synthetic Rules.

more notable improvement. For example, if the traffic is ruleset-intensive, then the detection load of the *DigenisNP* increases and the relative improvement is small. On the other hand, for traffic that requires less rules to be checked for every packet, the detection load of the *DigenisNP* will be minimal and the improvement will be greater.

We also repeat the experiment on a PC with a slower Pentium III processor at 1.13 GHz and the same PCI bus characteristics and Ethernet cards. The results (Figure 5.6) show that the improvement is smaller compared to the faster machine. When we examine more carefully the results, we observe that while *user time* doubles, the *system time* increases only by 30%. This is because *user time* is mainly the time spent for content search and header matching, which are processor intensive tasks. On the contrary, *system time* is dominated by the time spent for copying the packet from main memory, over the PCI bus, to the output network interface, handling interrupts and control registers of the Ethernet device. As the speed of processors increases faster than the speed of PCI buses and SDRAM memories, we can argue that, as technology evolves, the effect of our enhancements will be even more pronounced – processors are already running at 3.4 GHz, so the previously reported improvement is in fact a conservative result.

All the above experiments are performed using the default rule-set of *Snort*. To further understand the correlation between the load of a *DigenisPC* and the P-CACK scheme improvement we also experiment

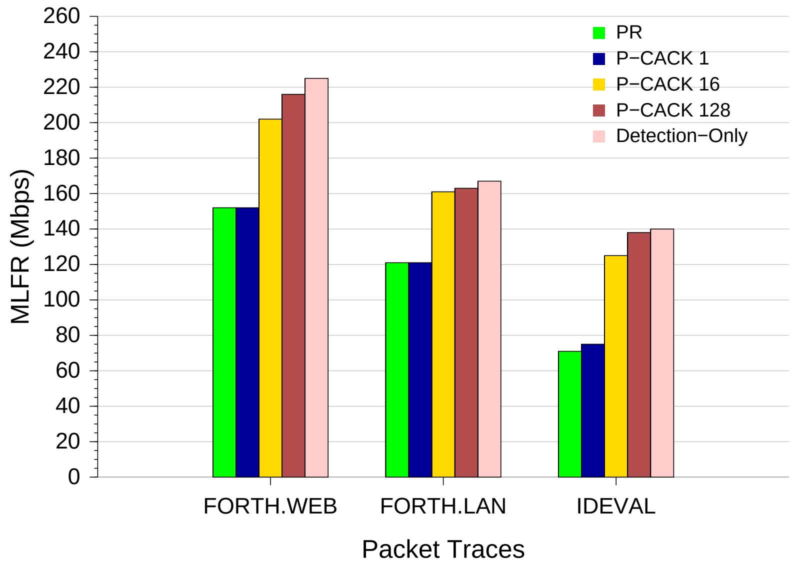


FIGURE 5.5: Maximum Loss Free Rate (MLFR) of a *DigenisPC* using Default Rule-Set.

alert tcp any any - any any (flags: S ; seq:47937 ; ID:48946 ; ttl:30280 ; fragbits: M ; content:"RPC overflow" ;)

TABLE 5.14: Synthetic Rule Example.

with variable, synthetic rule-sets. An example rule is shown in Table 5.14. The example rule instructs *Snort* to analyze all TCP packets searching for the string "RPC overflow" in the payload. Additionally, it checks whether the TCP SYN flag is set, the TCP sequence number is 47937, the IP ID is equal to 48946, the IP TTL equals 30280 and it is an IP fragment. If all these tests are true then the packet contains an attack. Similarly to the previous experiment, *Snort* reads traffic from a packet trace and send packets over a Gigabit Ethernet interface. The results are shown in Figure 5.4. We observe that as the number of rules increases the improvement of P-CACK scheme versus PR scheme decreases. In other words, as detection load increases, improvement decreases.

Another interesting point is that the maximum relative improvement of P-CACK over PR is for small packets of 64 bytes. Small packets require less time for content matching (*user time*) and communication (*system time*) is the dominant cost factor. In addition, in the case of 64-byte packets, the bootleneck is not the processor, as in the case of larger packets, but the PCI bus. This is clearly shown in the experiments

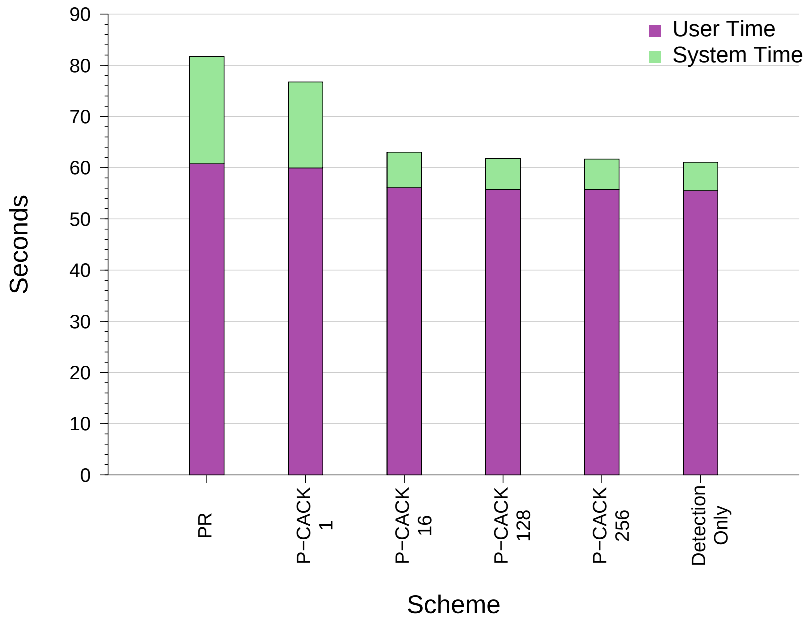


FIGURE 5.6: Processing Cost of the *DigenisPC*, with a Slower PC (**FORTH.WEB** Trace).

involving the **IDEVAL** traces, which contain many small packets for emulating certain types of attacks such as SYN flooding. For this trace, the P-CACK scheme is 3 times more efficient compared to the PR scheme. This is also a nice side effect of the P-CACK scheme, in that it makes the NIPS more robust against SYN flood attacks, given that such attacks consist of a big fraction of small packets.

5.1.4 Forwarding Latency of a *DigenisPC*

The highest portion of the latency imported by our NIPS is due to content matching on the *DigenisPCs*. This happens due to the fact that content matching is the single most expensive operation in every NIPS. To measure forwarding latency we use two hosts *A* and *B* with two Gigabit Ethernet interfaces each, *eth0* and *eth1*. We connect, the two interfaces of host *A* with the two interfaces of host *B* back-to-back. Everything that host *A* sends to network interface *eth0/eth1* is received by host *B* on network interface *eth0/eth1* and vice versa. Host *A* reads a trace from a file and sends traffic to host *B* (using *tcpreplay*[2]). Host *B* runs *Snort*, which receives packets from interface *eth0* and sends replies to interface *eth1*. Host *A* matches the packet sent time with the arrival of the reply and computes the latency.

Initially, we estimate the maximum loss free rate (MLFR) of a *DigenisPC* by replaying a packet trace and measuring the rate at which the *DigenisPC* started dropping packets (Figure 5.5). In this experiment we set the input packet buffer size to 16 Mbytes. We use MLFR to compute the latency that a *DigenisPC* imposes to analyzed packets when reaching its processing capacity.

In this experiment host *A* replays **FORTH.WEB** and **FORTH.LAN** traces at the maximum loss free rate **of each communication scheme**. We observe that there are packets that experience very high latency. To clarify this phenomenon we measure the time that *Snort* spends in content and header matching using the `rdtsc` [42] instruction of the Pentium IV. The results show that the peaks in time spent for content and header matching overlap with the peaks in latency. This means that, when the required per packet operations increase, so does the latency. A consequence of this property is that packets that require a significant amount of processing slow down other packets that do not. This is a form of head of line (HOL) blocking.

Figures 5.7, 5.9, 5.8, and 5.10 show the cumulative distribution function (CDF) for all ACK schemes when a *DigenisPC* receives traffic at the MLFR of **FORTH.WEB** and **FORTH.LAN** traces. We notice that, latency increases with the P-CACK factor. An interesting observation is that the graph is heavy tailed, meaning that while most of the packets experience low latency, 5% of the packets exhibit very high latency. These are packets that were received from *DigenisPCs* while it had a temporary excess load. This may happen because, for example, some packets require too many rules to be checked. If too many such packets are received back-to-back, the system reaches (or exceeds) its capacity and the latency increases considerably.

We repeat the previous experiment for traffic rates of 20%, 40% , 60%, and 80% of the MLFR (**FORTH.WEB** trace). The results are presented on Figures 5.11, 5.12, 5.13, and 5.14. Figure 5.11 shows that for the case of PR and P-CACK 1 schemes, the latency is below 2 milliseconds. For the case of P-CACK 16 the latency is below 2 milliseconds for 95% of the packets and only 5% of the packets experiences latency greater that 17 milliseconds. Regarding the P-CACK 128 scheme, we observe that the 65% of the packets experience latency less than 3 milliseconds, while the rest 35% experience latency above 17 milliseconds. Similarly, for the case of P-CACK 256 scheme we see that only 35% of the packets experience latency less that 3 milliseconds. We conclude that for the case of a P-CACK factor greater than 128, while there are packets that experience very low latency, there are also packets that experience unacceptably high latency for such a low traffic rate. This is a natural outcome of the way ACKs function. For instance, suppose that a packet reaches a *DigenisPC* while the *DigenisPC* has

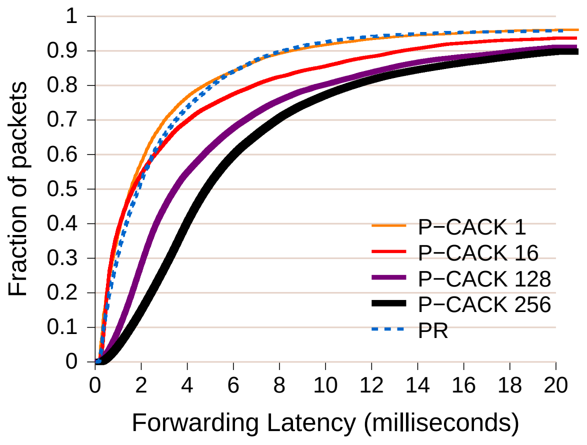


FIGURE 5.7: CDF for Latency at MLFR (**FORTH.WEB**) – 0 - 20 msec.

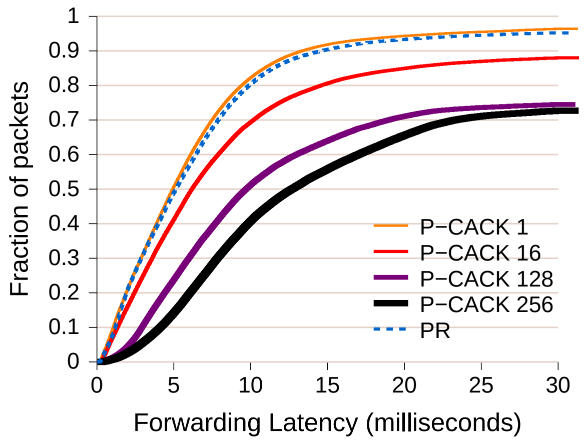


FIGURE 5.8: CDF for Latency at MLFR (**FORTH.LAN**) – 0 - 30 msec.

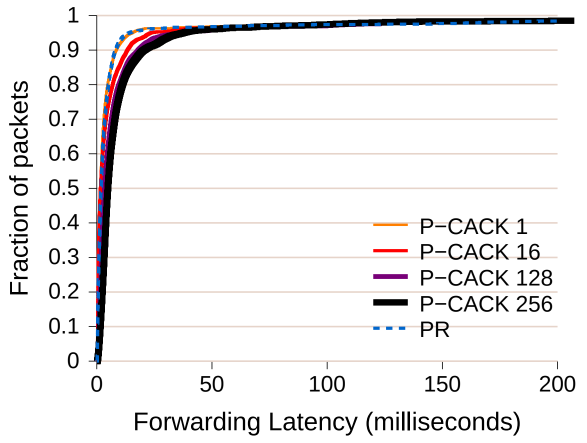


FIGURE 5.9: CDF for Latency at MLFR (**FORTH.WEB**) – 0 - 200 msec.

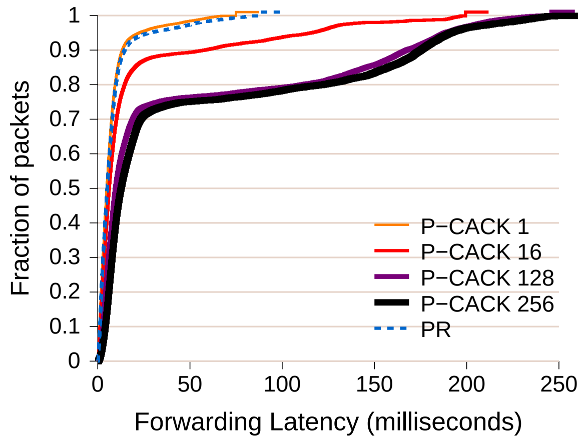


FIGURE 5.10: CDF for Latency at MLFR (**FORTH.LAN**) – 0 - 250 msec.

just send to *DigenisNP* an ACK and is configured to use a P-CACK factor of 256. Then this ”unlucky” packet has to wait for other 255 packets to be analyzed in order to be forwarded from *DigenisNP*, and as a consequence, it will experience very high latency. On the contrary, if a packet arrives on a *DigenisPC* just before an ACK is send to *DigenisNP*, this packet will experience very low latency. Conclusively, as the P-CACK factor increases, so does the probability that a packet will be unlucky and will suffer a high latency if the *DigenisPC* is receiving traffic at low rates.

Next, we notice that the difference in latency between the different ACK schemes fades away as the traffic rates increase. The reason is that the number of packets received per second increases and so the

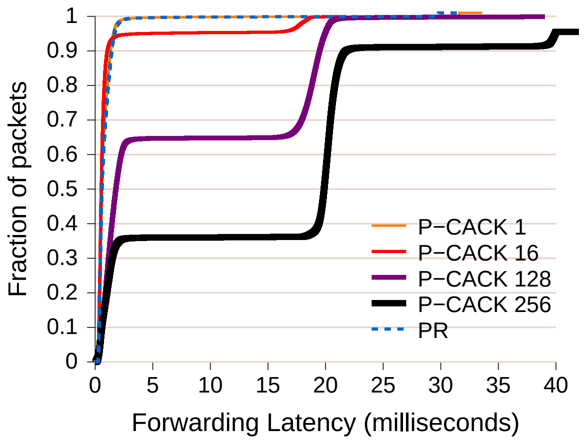


FIGURE 5.11: CDF for Latency at 20% of MLFR.

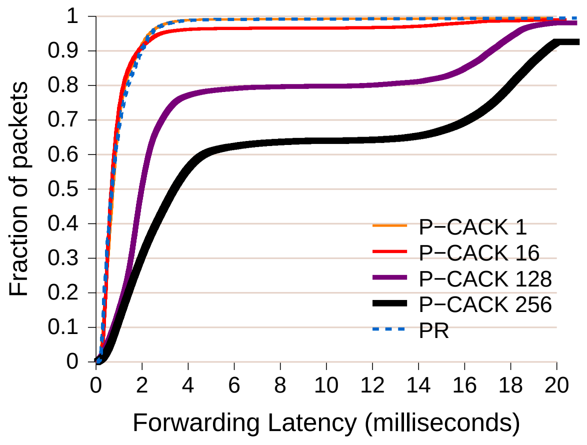


FIGURE 5.12: CDF for Latency at 40% of MLFR.

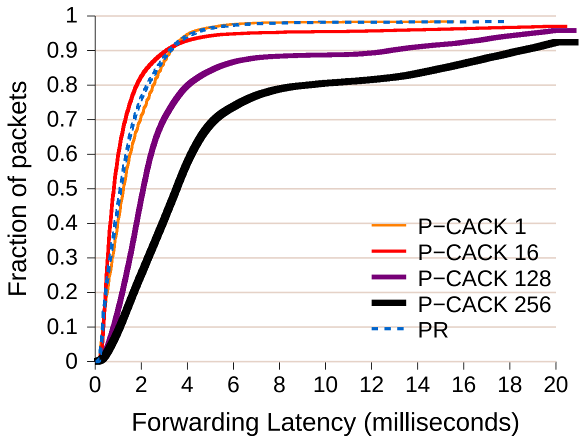


FIGURE 5.13: CDF for Latency at 60% of MLFR.

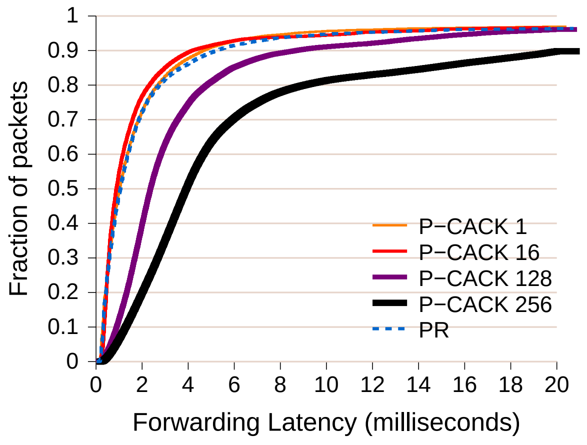


FIGURE 5.14: CDF for Latency at 80% of MLFR.

”big” P-CACK packet fills faster, resulting in lower latency. Thus, the P-CACK schemes with factors greater than 16 is better suited for *DigenisPCs* working at their processing capacity.

Finally, we measure the latency of the PR and the P-CACK schemes when the *DigenisPC* receives traffic at the MLFR of the PR scheme. The results are shown in Figure 5.15. We see that the latency of the PR scheme is worst than the latency of the P-CACK 1 and P-CACK 16 schemes. Thus, the reduction of the load that the P-CACK schemes achieve for these two P-CACK factors decreases the latency imported on the packets.

5.1.5 Forwarding Latency of *DigenisNP*

We argue that the overall latency that a packet experiences by our NIPS is due to the processing of *DigenisPCs* and not the forwarding of *DigenisNP*. Also, the cycles spent from *DigenisNP* to forward a

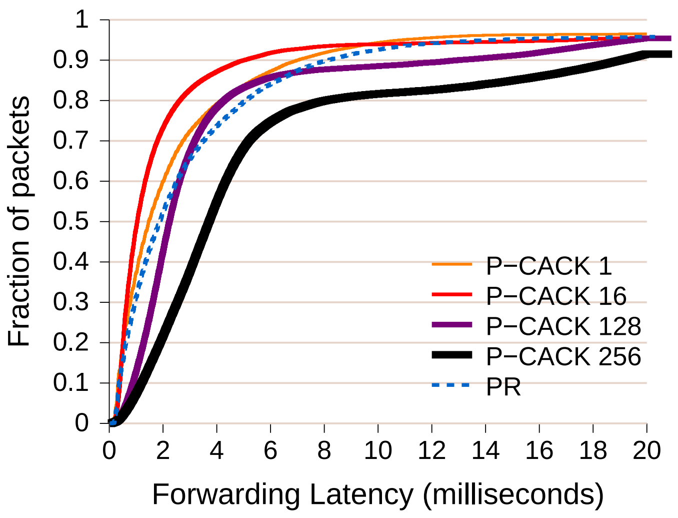


FIGURE 5.15: CDF for Latency for all ACK Schemes at the MLFR of the PR Scheme.

packet from the input port to an output port depend only on the packet length. This means that practically all packets experience almost the same latency.

5.1.6 Memory Requirements

There is a direct relationship between the latency imported by *DigenisPCs* and the required memory on *DigenisNP*. *DigenisNP* needs memory to save incoming packets until they are acknowledged by *DigenisPCs*. The amount of memory *DigenisNP* needs depends on the highest possible latency that our NIPS will tolerate. If we set this value in a reasonable value, for example, 200 milliseconds, then according to the fact that our NIPS analyzes traffic at 800 Mbit/s, the required memory is approximately 20 Mbytes. This means that the circular buffer of the *IXP1200* must be at least 20 Mbytes.

5.2 Evaluation of the Plug-in for Reducing Redundant Packet Inspection

In this Section we focus on the specific enhancement of packet caching. In particular, we compare the performance of *DigenisPC* with and without the packet caching optimization technique.

5.2.1 Experimental Environment

Experimental Environment for the *DigenisNP*

The performance of the configurations running on the *IXP1200* is measured using the Developer Workbench with the same configuration as in the previous plug-in. Moreover, we configure *DigenisNP* to store the fingerprints of the packet in the SDRAM. We also examine the performance of *Digenis* using SRAM as the storage area for fingerprints. In both cases, *Digenis* has the same performance. Moreover, we must mention that this is not a complete implementation of a packet cache. In particular, the fingerprint generation procedure we use, is in fact a lightweight fingerprint generation that probably produces hash collisions. A more serious approach would require a more demanding fingerprint computation procedure.

Experimental Environment for the *DigenisPCs*

We use a 2.4 GHz Pentium IV Xeon processor with hyper-threading disabled. The PC is equipped with 512 Mbytes of SDRAM memory at 133 MHz. The PCI bus was 64-bit wide clocked at 66 MHz. The host operating system is Linux (kernel version 2.4.20, Red-Hat 9.0). The Gigabit Ethernet card we use is Intel PRO/1000 MT Dual Port Server Adapter [14].

The software running on the PCs is a modified *Snort* version 2.2.0 compiled with gcc version 3.2.2. We turn off all the preprocessors of *Snort*. Unless noted otherwise, *Snort* is configured with the default rule-set. Furthermore, during the measurements we do not use the SHA-1 algorithm a more lightweight. Although, this algorithm is not as secure as the SHA-1, it did not generate collisions during the evaluation.

Packet Traces

During the evaluation of *Digenis* we use an **IDEVAL** trace different than the one used in the previous plug-in. Additionally, we use the **NASA.SYNTHETIC** packet trace which is a synthetic trace created with the following procedure. We download a log from a web server that was operating at NASA Kennedy Space Center in Florida and replay it in an Apache web server that we set up at **ICS-FORTH**. We use a client PC to send all the requests contained in the log to a PC running an Apache web server. We monitor the link with the HTTP requests to and responses from the web server and create a packet trace. Although this is not the same as real packet traces captured at links close to a real web server, it is the only approach we can use, given that privacy issues do not permit us to have access to traces with real packet payload. The characteristics of **NASA.SYNTHETIC** and **IDEVAL** traces are shown in Tables 5.15, 5.17, and 5.16.

Trace	Total Packets	Total MBytes	Average Packet Size (bytes)
NASA.SYNTHETIC	2853238	2101	736
IDEVAL	2855200	619	216

TABLE 5.15: Characteristics of the Packet Traces.

Packet Size	Percentage (%)
$size \leq 64$	70
$64 < size \leq 128$	11
$128 < size \leq 256$	5
$256 < size \leq 512$	3
$512 < size \leq 1024$	1
$1024 < size \leq 1518$	9

TABLE 5.16: Packet Size Distribution for **IDEVAL** Trace.

Packet Size	Percentage (%)
$size \leq 64$	0
$64 < size \leq 128$	50
$128 < size \leq 256$	3
$256 < size \leq 512$	0
$512 < size \leq 1024$	1
$1024 < size \leq 1518$	46

TABLE 5.17: Packet Size Distribution for **NASA.SYNTHETIC** Trace.

5.2.2 *DUCache* Hit Ratio

DUCache Stores Packets

With this experiment we aim to find the correlation between the size of the *DUCache* (in packets) and the hit ratio of the *DUCache*. To figure out this tradeoff we instruct *Snort* to process traffic from a trace for a varying number of *DUCache* sizes. We measure both the packet hit ratio and byte hit ratio. Besides, we

Cache Size	Packet Hit Ratio	Byte Hit Ratio	Packet Hit Ratio (w/o zero payload packets)	Payload Byte Hit Ratio
1 K	35.1	67.9	69.5	71.2
10 K	44.3	84.1	87.5	88.1
100 K	47.7	90.4	94.3	94.6
1 M	48.3	91.4	95.5	95.7
10 M	48.4	91.5	95.6	95.8

TABLE 5.18: *DUCache* Hit Ratio for **NASA.SYNTHETIC** Trace (*DU* is Packet).

Cache Size	Packet Hit Ratio	Byte Hit Ratio	Packet Hit Ratio (w/o zero payload packets)	Payload Byte Hit Ratio
1 K	5.7	7.7	10.6	8.6
10 K	9	16.8	16.8	19.8
100 K	12.2	27	22.8	32.6
1 M	19.9	48.4	37	59
10 M	22.1	54.5	41.2	66.5

TABLE 5.19: *DUCache* Hit Ratio for **IDEVAL** Trace (*DU* is Packet).

measure the packet hit ratio in the case that we exclude from the measurement the packets that have no payload and as a result can not be cached. Finally, we measure the byte hit ratio only when considering the payload of the packets. In other words, we do not count the size of the packet header. The results are shown in Tables 5.18 and 5.19. As expected, the hit ratio increases as the *DUCache* size increases.

In particular, in the case of **NASA.SYNTHETIC** trace that simulates a popular web server, the *DUCache* packet hit ratio without zero payload packets reaches 95.6%. This means that from the packets that have payload, only 4.4% are not in the *DUCache*. Finally, the packet hit ratio without zero payload packets in the case of **IDEVAL** trace is 41.2%. This hit rate is an artifact of the trace that is consisted of traffic that simulates intrusion attempts. A general observation is that the byte hit ratio is a little smaller than the payload byte hit ratio. This makes sense because the packets without payload are small packets and do not contribute much to the total traffic size.

Cache Size	*HDU* Hit Ratio	Byte Hit Ratio
1 K	61.5	60
10 K	82.3	82.6
100 K	90.3	91.5
1 M	91.7	92.9

TABLE 5.20: *DUCache* Hit Ratio for **NASA.SYNTHETIC** Trace (*DU* is *HDU*).

Cache Size	*HDU* Hit Ratio	Byte Hit Ratio
1 K	40.3	7.4
10 K	56.3	16.1
100 K	69.6	36.3
1 M	79.6	57.9

TABLE 5.21: *DUCache* Hit Ratio for **IDEVAL** Trace (*DU* is *HDU*).

DUCache **Stores** *HDUs*

We repeat the previous experiment but we cache *HDUs*[10] instead of packets[11]. As we observe the byte hit ratio is lower than the previous experiment. This makes sense because the bigger the *DU* the less probable it is to find two *DUs* that are the same.

5.2.3 Performance of *DigenisNP*

When *Digenis* is configured to run the packet caching technique on *DigenisNP*, *DigenisNP* ceases to handle only the headers of the packets and treat also the payload of the packets. As a consequence, it is not clear which is most demanding: traffic that contains small packets or traffic that contains large packets. To examine the utilization of the *IXP1200* we run *Digenis* with P-CACK factor equal to eight and with the packet caching plug-in enabled, for a varying number of packet sizes. Whereas *Digenis* sustains line speeds, we use as a metric for comparison the utilization of the microengines and the utilization of SRAM and SDRAM memories [12]. We run *Digenis* for two different *DUCache* hit ratios,

[10]*HDU* size ranges from 1 to 65535 bytes.

[11]We run *Snort* with the *stream4* preprocessor enabled.

[12]In particular, the utilization of the buses that drive to the SRAM and SDRAM memories.

Design	ME0	ME1	ME2	ME3	ME4	ME5	SDRAM	SRAM
Digenis (Hit Ratio 100%)	57	56.1	65.4	65.4	60.8	66.3	21.6	15.8
Digenis (Hit Ratio 0%)	58.2	56.4	69.2	69.3	69.6	65.5	41	32.5

TABLE 5.22: Utilization(%) of the Microengines, SDRAM and SRAM for 64-byte Packets.

Design	ME0	ME1	ME2	ME3	ME4	ME5	SDRAM	SRAM
Digenis (Hit Ratio 100%)	52.7	52.7	66.5	66.5	61.6	64.8	22.5	6.9
Digenis (Hit Ratio 0%)	51.1	51.0	70.7	70.7	71.5	64.4	45.9	15.5

TABLE 5.23: Utilization(%) of the Microengines, SDRAM and SRAM for 512-byte Packets.

Design	ME0	ME1	ME2	ME3	ME4	ME5	SDRAM	SRAM
Digenis (Hit Ratio 100%)	52.3	52.4	66.6	66.6	61.6	61.6	22.8	5.9
Digenis (Hit Ratio 0%)	51.2	51.4	71	71	71.5	64.4	46.6	14

TABLE 5.24: Utilization(%) of the Microengines, SDRAM and SRAM for 1518-byte Packets.

100% and 0%. Obviously, the second case is the most demanding. In Tables 5.22, 5.23, and 5.24 we present the utilization of the microengines and the utilization of the SRAM and SDRAM memories.

First, in the case of 64-byte packets, we observe that the utilization of the first two microengines has increased compared to the no packet caching case. This is because the code that is responsible for the generation of the fingerprints runs on these two microengines. Moreover, as we observe, in the case that the *DUCache* hit ratio is 100% the SDRAM and SRAM utilization has decreased. This makes sense, because if a packet is in the *DUCache, DigenisNP* neither forwards it to the *DigenisPCs* nor receives an ACK back, saving this way a significant number of memory transactions. In contrast, when the *DUCache* hit ratio is 0%, SDRAM utilization increases compared to the no packet caching case. In this case, all packets cause a *DUCache* miss and thus, we generate fingerprints and perform *DUCache* operations for no reason [13]. The same observation holds also for the measurements with 512- and 1518-byte packets.

Queue Entries	Percent	Cumulative Percent
0	59.5/39.8	59.5/39.8
1	16.2/19.5	75.8/59.3
2	10.8/14.3	86.6/73.7
3	6.6/10.6	93.2/84.3
4	3.8/7.6	96.9/91.9
5	2.1/5.3	99.1/97.2
6	0.8/2.2	99.8/99.5
7	0.2/0.5	100/100

TABLE 5.25: FBI Pull Queue Fullness Statistics (64-byte Packets).

Queue Entries	Percent	Cumulative Percent
0	67.7/65.7	67.7/65.7
1	18.2/19	85.8/84.7
2	7/7.7	92.9/92.4
3	3.6/3.6	96.4/96.1
4	1.4/1.7	97.8/97.7
5	1.1/1.3	99/99
6	0.8/0.8	99.8/99.8
7	0.2/0.2	100/100

TABLE 5.26: FBI Push Queue Fullness Statistics (64-byte Packets).

[a]

[a]The results are shown as – 100% hit ratio case / 0% hit ratio case.

Queue Entries	Percent	Cumulative Percent
0	73.9/50.9	73.9/50.9
1	19.4/23.9	93.2/74.7
2	4.4/11.9	97.6/86.6
3	1.3/6.3	99/92.9
4	0.7/3.7	99.6/96.6
5	0.3/2.2	99.9/98.8
6	0.1/1.0	100/99.8
7	0/0.2	100/100

TABLE 5.27: FBI Pull Queue Fullness Statistics (1518-byte Packets).

Queue Entries	Percent	Cumulative Percent
0	58.9/53.6	58.9/53.6
1	26.2/25.3	85.1/78.9
2	10.1/12.1	95.2/91
3	3.6/5.6	98.7/96.6
4	0.9/2.3	99.7/99
5	0.2/0.8	99.9/99.8
6	0.1/0.2	100/100

TABLE 5.28: FBI Push Queue Fullness Statistics (1518-byte Packets).

Queue Entries	Percent	Cumulative Percent
0	94.1/93.5	94.1/93.4
1	5.8/6.4	99.9/99.9
2	0.1/0.1	100/100

TABLE 5.29: Hash Queue Fullness Statistics (64-byte Packets).

[a]

[a]The results are shown as – 100% hit ratio case / 0% hit ratio case.

Queue Entries	Percent	Cumulative Percent
0	69/65.3	69/65.3
1	23.1/23.1	92.1/88.4
2	6.4/8.2	98.5/96.6
3	1.3/2.6	99.8/99.3
4	0.2/0.6	100/99.9
5	0/0.1	100/100

TABLE 5.30: Hash Queue Fullness Statistics (1518-byte Packets).

Utilization of the FBI Unit

As we have already mentioned, another possible bottleneck of the *IXP1200* is the FBI unit. Tables 5.25, 5.26, 5.27, and 5.28 show the number of entries of the FBI push/pull queues for *Digenis* and for 64- and 1518-byte packets (P-CACK factor is eight and cache hit ratio is 100% and 0%). When we use 64-byte packets, we observe that the FBI pull queue has more entries than in the case that we do not use packet caching, while the push queue has almost the same number of entries as in the case that we do not use packet caching. Also, we see that the FBI pull queue has increased utilization in the case that the *DUCache* hit ratio is 0%.

Regarding the 1518-byte packets, the results show that in the 0% *DUCache* hit ratio case, 92.9% of the time, the size of the pull queue is less than 50%, while 96.6% of the time, the size of the push queue is less than 50%. In addition, the utilization of the FBI pull queue increases as the packet size decreases, while the utilization of the push queue increases as the packet size increases.

Utilization of the Hash Unit

Tables 5.29 and 5.30 show the number of entries of the queue using 64- and 1518-byte packets (the P-CACK factor does not affect the result) and for 0 and 100% *DUCache* hit ratio. We observe that, when 64-byte packets are used the hash unit is less loaded compared to the case that 1518-byte packets are used. Also, with 64-byte packets, we observe that the utilization of the hash unit increases compared to the case that we do not use the packet caching technique. Regarding 1518-byte packets, the results show that in the worst case (0% hit ratio), 65.3% of the time the queue of the hash unit has no pending requests, 88.4% of the time the hash unit has one pending request, and only 11.6% of the time it has more than two pending requests. Thus, the utilization of the hash unit slightly increases compared to the case that the packet caching plug-in is disabled.

5.2.4 Performance of *DigenisPCs*

When the packet caching is applied to *DigenisNP*, the performance boost that *DigenisPCs* experience is greater than the case where packet caching is applied on *DigenisPCs* themselves. This is due to the fact that, when packet caching is applied on a *DigenisPC*, we have to take into account the processing cost of fingerprint generation, the lookup of this fingerprint, and the insertion in case of a *DUCache* miss.

DUCache Stores Packets

In this experiment we instruct *Snort* to process traffic from a trace with and without the packet caching technique and we measure the total processing time. The results for a varying *DUCache* size are shown in Tables 5.31 and 5.32. As we observe for the case of the **NASA.SYNTHETIC** trace, *Snort* with a *DUCache* size of 10 million packets run almost 4 times faster than unmodified *Snort*. Also, the processing time of *Snort* with *DUCache* is about 38% less than the processing cost of the unmodified *Snort* for the **IDEVAL** trace.

Additionally, in order to figure out the overhead that fingerprint generation and *DUCache* lookup/insertion induces, we perform the following experiment. We run *Snort* with the *DUCache* disabled, then with the fingerprint generation enabled and then with the fingerprint generation and the lookup/insertion enabled.

[13]Remember that the *DUCache* resides on the SDRAM. Additionally, we use SDRAM as a temproray storage area for fi ngerprint generation.

Snort	Total Processing Time (Seconds)
No *DUCache*	55.4
DUCache (1 K)	20.5
DUCache (1 M)	14.5
DUCache (10 M)	14.5

TABLE 5.31: Total Processing Time of *Snort* for **NASA.SYNTHETIC** Trace (*DU* is Packet).

Snort	Total Processing Time (Seconds)
No *DUCache*	16.8
DUCache (1 K)	17.2
DUCache (1 M)	12.7
DUCache (10 M)	12.1

TABLE 5.32: Total Processing Time of *Snort* for **IDEVAL** Trace (*DU* is Packet).

Trace	No *DUCache*	Fingerprints Enabled	Fingerprints and *DUCache* Operations Enabled (1M)
NASA.SYNTHETIC	55.4	61.2	61.6
IDEVAL	16.8	18.3	18.7

TABLE 5.33: Processing Cost (Seconds) of Fingerprint Generation and Cache Operations (*DU* is Packet).

In all these cases the packets are handed to the detection engine for further analysis. The results are presented on Table 5.33. We observe that the overhead of fingerprint generation and the *DUCache* operations is approximately 11%.

Next, we perform the following experiment. We modify *Snort* to dump every packet that is not found in the *DUCache* into a packet trace. We run *Snort* with a *DUCache* of 10 million packets using the **IDEVAL** and **NASA.SYNTHETIC** traces as inputs and, thus, generate the traces **IDEVAL**.uniq and **NASA.SYNTHETIC**.uniq. These new traces, contain packets of the old traces without the duplicated ones. Then, we instruct an unmodified *Snort* to process these newly created traces and measure the

Trace	Total Processing Time	Improvement
NASA.SYNTHETIC.uniq	5.4	10 times faster
IDEVAL.uniq	9.2	82% faster

TABLE 5.34: Processing Cost (Seconds) to Analyze All Packets in the Traces Containing Unique Packets.

Snort	Total Processing Time
No *DUCache*	108.3
DUCache (1 K)	86.3
DUCache (100 K)	71.5
DUCache (1 M)	70.7

TABLE 5.35: Total Processing Time of *Snort* for **NASA.SYNTHETIC** Trace (*DU* is *HDU*).

Snort	Total Processing Time
No *DUCache*	30.2
DUCache (1 K)	30.4
DUCache (100 K)	27.7
DUCache (1 M)	26.2

TABLE 5.36: Total Processing Time of *Snort* for **IDEVAL** Trace (*DU* is *HDU*).

processing time of *Snort*[14]. The results are shown in Table 5.34. The results are similar to the case where *DigenisNP* runs the packet caching technique. Thus, we observe that *Snort* runs from 82% to 10 times faster compared to the processing time required when we use the original traces.

DUCache **Stores** *HDUs*

We repeat the experiment regarding the total processing time of *Snort* but we cache *HDUs* instead of packets. The first observation is that the total processing time is higher than the previous experiment. This extra processing time is the cost of reconstructing the TCP connections into *HDUs*. This time, in the case of **NASA.SYNTHETIC** trace the improvement is much lower, only 53%. This is caused

[14]*Snort* didn't had the *stream4* preprocessor activated.

Trace	No *DUCache*	Fingerprints Enabled	Fingerprints and *DUCache* Operations Enabled (1M)
NASA.SYNTHETIC	108.3	113.8	114.4
IDEVAL	30.2	31.6	32.6

TABLE 5.37: Processing Cost (Seconds) of Fingerprint Generation and Cache Operations (*DU* is *HDU*).

by the fact that the TCP connection reconstruction procedure that takes place irrespective of the packet caching technique, consumes a big portion of the total processing time of *Snort*. Thus, packet caching has a smaller effect. The same observations holds for the **IDEVAL** as well trace where improvement decreases from 38% to 15.2%.

Finally, in order to figure out the overhead that fingerprint generation and *DUCache* lookup/insertion induces, we perform the following experiment. We run *Snort* with the *DUCache* disabled, then with fingerprint generation enabled, and finally with fingerprint generation and lookup/insertion enabled. In all cases *HDUs* are handed to the detection engine for further analysis. The results are presented on Table 5.37. We observe that the overhead of fingerprints generation and the *DUCache* operations varies between 6-8%.

6
Related work

Using load-balancing for scaling network intrusion detection has been studied in [23]. The authors propose a three-tier architecture for scaling stateful intrusion detection. They describe a partioning approach that supports in-depth, stateful intrusion detection on high-speed links. The traffic is captured by a traffic *scatterer*, which equally distributes packets to a set of traffic *slicers*, in a round-robin fashion. Subsequently, the traffic *slicers* are connected through a *switch* with a set of intrusion detection engines. The slicers examine packets for determining a suitable set of detection engines for final processing. The decision about which detection engine will analyze the packet, is based on rules describing the attack contexts to which a packet may belong. Another way to look into this architecture, is that the definition of a flow is not fixed, as in our architecture, but it is shaped by the rules running on the slicers. The main focus of the work is, therefore, in preserving detection semantics in a generalized model of intrusion detection, assuming different types of detection such as statistical methods, anomaly detection, and misuse detection. On the other hand, we focus only on misuse detection methods. This enables us to use a fixed definition of flows, as decribed in Chapter 2, after we verify that there are no attack patterns that split over multiple flows (with the exception discussed in Chapter 3). This way, we can replace the complex rules that slicers run with a flow-preserving load balancer. Thus, we can state that *DigenisNP*, in reality, combines the functionality of the scatterer, the slicers, and the switch, in the case that these components were to be used in misuse detection systems. This approach allow us to lower considerably the cost of the whole system[1]. In addition, we investigate ways on how to offload the detection engines by rethinking the mapping of operations to the various components of the system.

Charitakis et al. [7] implement a system with a splitter distributing traffic to a number of NIDSes[2]. The splitter is an *IXP1200* network processor board[3] that implements operations on the traffic stream with the goal of reducing the traffic on the NIDSes. They propose early filtering and locality buffering as two techniques to improve the performance of the NIDSes. The early filtering technique, moves part of the functionality of the NIDSes to the *IXP1200*. In particular, they move the detection heuristics of the NIDSes that do only packet header analysis, and therefore, are cheap to perform, to the *IXP1200*. Moreover, the *IXP1200* is responsible for reordering the packets passing through, in order to improve memory access locality on the NIDSes. The architecture we propose is similar to the one proposed in [7], but important differences exist. First of all both techniques proposed in [7] are not applicable to our system. In the first technique, the *IXP1200* drops all the packets that contain no payload after it has inspected them for attacks. However, doing so in the case of *Digenis* is not possible due to the fact that many TCP acknowledgments are encapsulated into packets with no payload. These packets are required by *DigenisPCs* in order to reconstruct the original TCP connection before applying the detection heuristics. Similarly, the locality buffering technique, while it can be used in a NIDS, it can not be used in a NIPS because it introduces unacceptable latency. Moreover, recall that a NIPS is operating in-line and, as a consequence, any reordering of the packets is noticed by the end-host and may affect negatively the retransmission mechanism of the TCP, the most frequently used transport protocol on the Internet. Consequently, in contrast to [7], we have to take into account parameters that do not exist in a NIDS, but are very important when we consider a NIPS. Finally, the packet caching technique we propose can also be used in [7] without any modification.

[1]The *IXP1200EEB* costs even less than the switch used in [23]

[2]They use *Snort* as a NIDS.

[3]Manufactured by *Radisys* [33].

Other research efforts recognize the issue of flexibility and implement NIDSes in reconfigurable hardware. However, programming hardware is much more difficult than programming NPs. Schuehler et al. [38] describe an architecture for a hardware based, TCP/IP content scaning system, capable of performing complete, stateful, payload inspections on eight million TCP flows at 2.5 Gbit/s. They use a hardware circuit that combined a TCP processing engine, a per flow state store and a payload scanning engine. Necker et al. [27] implement portions of the functionality of a state-of-the-art NIDS in reconfigurable hardware. Specifically, they implement TCP stream reconstruction and state tracking in a reconfigurable network interface based on Xilinx Virtex technology. Li et al. [24] use FPGA-based reconfigurable hardware to implement the detection engine of *Snort* v1.8.7. Their architecture should be able to monitor networks with a speed up to 2.68 Gbit/s. Clark et al. [9] present a distributed architecture where the intrusion detection and prevention is accomplished on the end-hosts. The stateful inspection is accelerated using the *IXP1200* network processor and the Xilinx Virtex FPGA. They describe how they map the operations of *Snort* in their hardware and they demonstrate that their hardware is capable of working at 100 Mbit/s. Dharmapurikar et al. [12] present a technique based on Bloom filters [4] for detecting predefined signatures in the packet payload. They implement the Bloom filters in hardware and use an ordinary string search algorithm for eliminating false positives produced by the filters. They state that their system is capable of scanning 10000 strings at 2.4 Gbit/s.

A number of vendors have used NPs to accelerate intrusion detection. Cisco uses *IXP1200* on *Cisco Catalyst 6500 Series IDS Module (IDSM-2)* [8] which is a platform capable of performing intrusion detection at 600 Mbit/s with 450-byte packets. This system supports up to 4000 TCP connections per second (new arrivals) and up to 500.000 concurrent connections. Consystant [11] claims that has implemented *Snort* on the *IXP2400* network processor but details on the performance of this design are not available.

A number of vendors claim to have NIPSes that can operate at high speed links. For example, ISS offers *Proventia G200* [15], a system designed for 200 Mbit/s networks. This device uses a software-based detection engine on an Intel platform. NetScreen provides *IDP 500* [28] designed for 500 Mbit/s networks. This sensor is a hardware appliance that runs the Linux-based *IDP Sensor* software, it is based on the Dell PowerEdge 1750 hardware platform with dual-Pentium IV processors and 4GB RAM. McAffe demonstrates *IntruShield 4000 Sensor (I-4000)* [29] and claims real-time prevention at speeds up to 2 Gbit/s. In order to be able to reach that speed, *I-4000* uses custom hardware for capturing packets and effectively detecting and blocking intrusions. TippingPoint uses custom-designed high-speed security processors on *UnityOne 2400* [43] and claims aggregate throughput of 2 Gbit/s. Top Layer presents *Attack Mitigator IPS 2400* [44], a combination of multiple *Attack Mitigator IPS 1000* and load balancer units capable of analyzing 1 Gbit/s networks. Incoming traffic in evenly spread by a load balancer to four *Attack Mitigator IPS 1000* devices and from there to a second load balancer which forwards the traffic to its destination.

Finally, with respect to packet caching, Santos et al. [36] investigat whether there is redundancy in the packets flowing through a link and propose a technique for increasing effective link bandwidth by suppresing replicated packets. In particular, they find significant replication (24%) of the packets in the case of HTTP traffic. The traffic they investigate is captured at the exchange point between MIT and the Internet and consists of a couple of laboratories. Spring et al. [41] present a technique for identifying replicated data but use a smaller granularity than Santos et al. In particular, they propose a technique for identifying repeated byte ranges between packets to avoid retransmitting the redundant data. They find about 30% redundancy in the case of HTTP protocol traffic.

7 Conclusions

This thesis demonstrates that a high-performance, low-cost, flexible and scalable NIPS can be build using network processors and PCs. We show this by presenting a novel architecture, key implementation methods and performance evaluations.

The main contribution of this thesis is to demonstrate that it is feasible to build a high-performance, low-cost, flexible and scalable NIPS using commercially available network processors and PCs. To show this, we (1) describe an achitecture, called *Digenis*, that is flexible, scalable, cheap, and handles high-speed links; (2) present implementation techniques to realize this architecture; and (3) present the allocation of operations to components and the trade-offs faced during the design and prototyping of such systems (Figure 7.1).

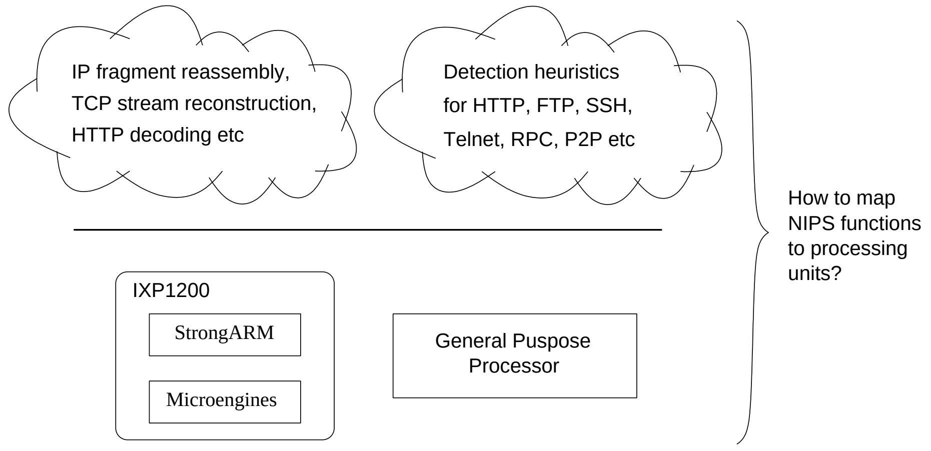


FIGURE 7.1: Mapping of Functions to Processing Units.

7.1 Future Work

There are several directions we can extend the work presented in this thesis. We present these directions in the sections that follow.

7.1.1 More Sophisticated Reduction of Redundant Inspection

We would like to investigate the possibility of further reducing redundant inspection by trying to find redundancy not in the packet or *HDU* level but in a smaller scale. In particular, we would like to examine if it is possible to find redundancy in parts of the *DUs*. For instance, if two *DUs* differ only in 10 contiguous bytes of their payloads, we could take advantage of this similarity and avoid redundant inspection of the portion of the *DUs* that is the same.

The challenge here is to cache portions of the packets in order to be able to identify commonality and keep the cache a reasonable size. As an example, one could cache the hashes of all aligned 256 bytes substrings of the packets. To check for similarity, the algorithm would compute only the fingerprints of the aligned 256 substrings and would check for membership. Unfortunately, two packets that are almost the same but one of them has an extra byte at the start of the payload would shift all substring boundaries, change the fingerprints of all substrings and prevent any potential exploitation of the redundancy. An alternative solution is to cache the fingerprints of all overlapping 256 bytes substrings at all offsets. Such an algorithm would require a cache with enormous size (almost one cache entry for per byte of packet payload). Even if the size is tolerable, the lookup operations in such a huge cache will degradate performance.

We have implemented a technique that is used in the LBFS filesystem [26]. This technique considers only non-overlapping substrings and is resilient to shifted substring by setting substring boundaries based on substring contents, rather than on position within a packet. To divide a packet into substrings we examine every overlapping N_1-byte window of the packet and with probability 2^{-N_2} over the contents of each window we consider it to be the end of a substring. In order to select these boundary regions (breakpoints) we use Rabin fingeprints [5]. Rabin fingeprints are efficient to compute on a sliding window on a packet. When the low-order N_2 bits of the fingeprint of a window equal a predefined value, the window constitutes a breakpoint.

We would like to experiment with the values N_1 and N_2 for traces containing real packet payloads. Moreover, we would like to investigate ways to take advantage of the possibly increased redundancy in a way that prevents evasion attacks. Preliminary results demonstrate that the benefit from this technique is greater than the less fine-grained packet caching technique.

7.1.2 Structure of the NIDS

During the design and implementation of *Digenis* it became clear that the structure of *Snort* does not map easily to a multilevel processing hierarchy such as *Digenis*. While *Snort* is the state-of-the-art NIDS, used by many security experts and the scientific community, we believe it has been designed with the architecture of a PC in mind. Specifically, it is very difficult (if not impossible) to split *Snort* into independent parts that could be easily implemented on different processors. This is the reason that, both plug-ins we have designed, perform only system level optiomizations. For these reasons, we would like to investigate the possibility to use more fine-grained NIDSes, such as *Bro*.

Bro is a protocol-based network intrusion detection system. The detection engine of *Bro* instead of performing string searching for every analyzed packet, like *Snort* does, it creates events based on the monitored traffic. After an event is generated, *Bro* runs any event handler that is associated with this event. The event handler is written is the *Bro* language and is interpreted by the *Bro* interpreter. The ”programs” or ”scripts” written in the *Bro*, in contrast to *Snort*, identify attacks by looking for operations not supported by the network or higher-level protocol or deviate from the administrator’s policy.

We observe that, as *Snort* evolves, it is moving towards an architecture similar to *Bro*. If someone examines the number of detection heuristics added to *Snort* in the form of signatures and in the form of preprocessors (which is a form of *Bro*-like programs) then he will notice that the number of lines of C code for the preprocessors are significantly more than the signatures added. Moreover, many of the signatures added do not perform string searching but protocol decoding. This is better understood with an example. Figure 7.2 shows some of the signatures of *Snort* for RPC traffic.

The first of these signatures instructs *Snort* to apply this detection heuristic for every UDP packet. The detection heuristic searches for content ”—00 01 86 A0—” starting at the 12th byte of the payload of the packet until the 16th byte. However, notice that the content that we search for, is 4 bytes. Thus, in reality we do not perform a string searching but a string comparison. The same observation holds for other signatures as well.

7.1.3 HTTP Protocol Decoding on *DigenisNP*

Suppose that we are an ISP that hosts a popular web site and we want to protect our web servers from possible compromises. Because we do not want to protect the web clients, we do not care if the web servers distribute malicious content. It is the responsibility of the clients to secure themselves. Thus, it is possible to inspect for intrusions only the part of the TCP stream that contains the HTTP requests of the clients and the HTTP response of the server. We can safely discard the portion of the stream that carries the actual HTTP content (e.g HTML files). Although, someone may naively believe that we could also skip the check of the HTTP response, this is not the case. This occurs because there are situations where, while *Digenis* has missed an attack in the first place, it can infer that a machine is compromised by the responses that a machine generates.

This optimization is already present in *Snort*. However, the benefits from performing this optimization on *DigenisNP*, would be more pronounced. In this case, *DigenisNP* would forward immediately the packets that are part of HTTP content. Consider that 95% of the HTTP traffic is server-side traffic and 95% of the server traffic is HTTP content. This means that *DigenisNP* would immediately forward 90% of the HTTP traffic. The first benefit would be that the latency experienced by the packets would be very low. The second benefit is that we could save the 90% of the processing time that *Snort* spents on packet reception (we have already minimized packet transmission). Previous research states that the receive processing on *Snort* accounts for 30% of total CPU time [13].

7.1.4 Separating Server-Side and Client-Side Traffic

Snort classifies the detection heuristics into server-side and client-side. The server-side heuristics are applied only to traffic originating from servers and the client-side heuristics are applied only to traffic originating from clients (and destined to servers). However, there are more client-side heuristics than server-side heuristics. For example, in *Snort* version 2.2.0, there are 2040 detection heuristics and only 98 of them are server-side heuristics. Additionally, as we mentioned above, the server-side traffic occupies the highest portion of the links. We believe that if we could create a small detection engine that will execute on *DigenisNP*, we would have signifant benefits. However, this solution requires from *DigenisNP* to perform IP fragment reassembly and TCP stream reconstruction in order to prevent evasion attacks.

As we have mentioned in Chapter 4, in the case of the *IXP1200*, the IP fragment reassembly task can be done on the StrongARM because in general a small percentage of the total traffic is fragmented. However, a respectable portion of TCP stream reconstruction functionallity must be performed by the microengines. A fact that helps us a lot is that previous research [3, 31] states that 85% of the total TCP traffic is in order. Thus, we could program microengines to check if the input packet is in order, and in case it is not, to enqueue it in the StrongARM for further processing. Then, the StongARM takes the responsibility to put the packet in the right place into the reconstructed stream.

```
alert udp $EXTERNAL_NET any -> $HOME_NET 111    \
(content:"|00 01 86 A0|"; depth:4; offset:12;   \
 content:"|00 00 00 05|"; within:4; distance:4; \
 content:"|00 00 00 00|"; depth:4; offset:4;    \
 )

alert udp $EXTERNAL_NET any -> $HOME_NET 111    \
(content:"|00 01 86 A0|"; depth:4; offset:12;   \
 content:"|00 00 00 04|"; within:4; distance:4; \
 content:"|00 00 00 00|"; depth:4; offset:4;    \
 )

alert tcp $EXTERNAL_NET any -> $HOME_NET 111    \
(flow:to_server,established;                    \
 content:"|00 01 86 A0|"; depth:4; offset:16;   \
 content:"|00 00 00 04|"; within:4; distance:4; \
 content:"|00 00 00 00|"; depth:4; offset:8;    \
 )

alert tcp $EXTERNAL_NET any -> $HOME_NET 111    \
(flow:to_server,established;                    \
 content:"|00 01 86 A0|"; depth:4; offset:16;   \
 content:"|00 00 00 01|"; within:4; distance:4; \
 content:"|00 00 00 00|"; depth:4; offset:8;    \
 )

alert udp $EXTERNAL_NET any -> $HOME_NET 111    \
(content:"|00 01 86 A0|"; depth:4; offset:12;   \
 content:"|00 00 00 01|"; within:4; distance:4; \
 content:"|00 00 00 00|"; depth:4; offset:4;    \
)
```

FIGURE 7.2: Signatures of *Snort* for RPC Traffi c.

8

Appendix A

8.1 The *IXP1200* Network Processor Overview

The *IXP1200* is a device that is ideally suited for network elements. It intergrates seven microprocessor cores, with twenty four independent threads of execution on a single chip (Figure 8.1). The *IXP1200* consists of a StrongARM general purpose processor and six microengines. The microengines are RISC processors with an entirely new instruction set, well suited to high-speed data manipulation and movement. Each microengine intergrates four independent hardware threads with accompanying logic to manage them and a large register set. The *IXP1200* is able to interface with PCI, SRAM, SDRAM and IX buses. The IX bus is the high speed data flow interface to the *IXP1200* and is used for interconnecting IXP1200 chips and for packet I/O. There are also 4 Kbytes of on-chip RAM, called scratchpad, I/O buffers and on-chip queues for optimizing external memory accesses (Figure 8.2).

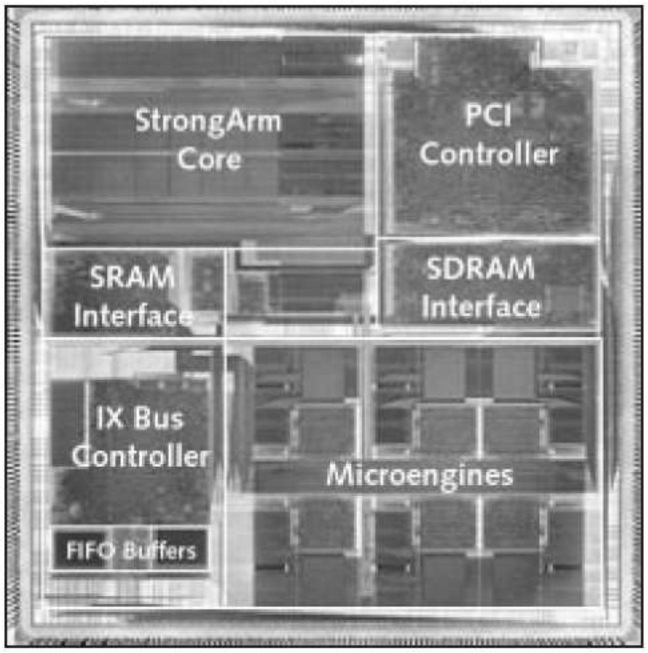


FIGURE 8.1: The IXP1200 chip (*Taken from Microprocessor Report, Volume 13, Number 12).*

8.1.1 Microengines

These compact RISC cores, are fully programmable 32-bit engines with a five stage execution pipeline, a large register set and hardware multithreading. Hardware multithread support allows four seperate programs to share execution time on a microengine. The overhead associated with switching contexts is a maximum of one cycle (however this overhead can be eliminated with a deferred instruction).

8.1.2 An Immense Amount of Registers

Each microengine has 128 general purpose registers (GPRs) and 128 transfer registers and a set of control and status registers (CSRs). All are 32-bits wide and are single-ported to reduce circuit complexity. From the 128 transfer registers, half of them are SDRAM transfer registers and the other half of them are SRAM transfer registers. These transfer registers are further divided into 32 read and 32 write transfer registers.

All accesses to SRAM and SDRAM use the transfer registers. ALU operations use the GPRs but in contrast to other architectures, transfer registers can also be used as source or destination operands. Thus, it is not required to copy values from transfer registers to GPRs to perform common ALU operations

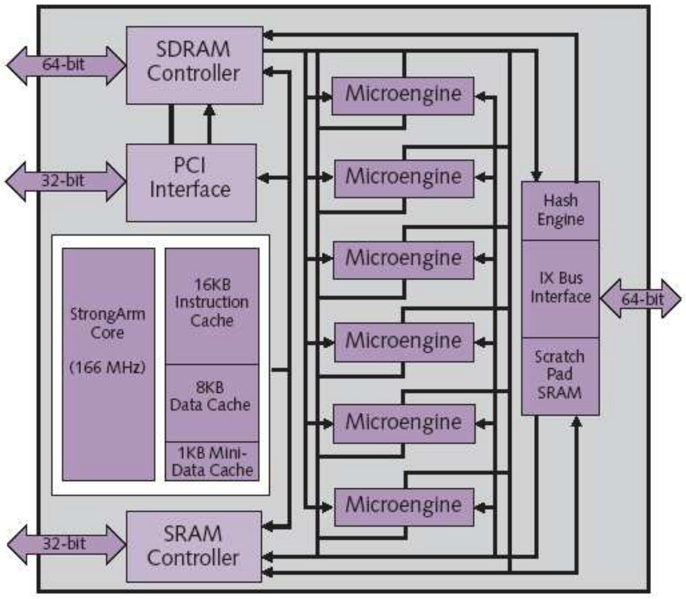


FIGURE 8.2: The IXP1200 Intergrates Seven Microprocessor Cores on One Chip (*Taken from Microprocessor Report, Volume 13, Number 12)*.

making the programming task easier.

As you may have observed there are no transfer registers for the scratchpad memory. That is because the *IXP1200* uses the SRAM instruction (and consequently SRAM registers) not only for accessing SRAM but also to access the scratchpad memory, the CSRs and the receive/transmit buffers for packet I/O.

8.1.3 Hardware Multithreading

One of the most fascinating feautures of the IXP1200 is the *hardware multithreading*. Hardware multithreading gives the processor the opportunity to hide the long latencies of memory accesses by switching contexts with other threads. This context switching is performed by special hardware logic and requires minimal programming effort.

For better understanding of this feature, we give a simple example. The microengine instruction set has an instruction for reading from off-chip SDRAM. The hardware multithreading feauture can be invoked by appending an optional token to the instruction:

```
SDRAM [read, xfer2, addr_sdram, 2], ctx_swap
```

In a nutshell, this tells a microengine to read two quad-words (four 32-bit words) from the SDRAM address *addr_sdram* and store the values in four contiguous registers starting at register *$$xfer2*. The final token, *ctx_swap*, permits the thread arbitrer to pass control to another thread in this microengine until the data from memory are transfered to the destination registers. In effect, this thread will block in this line until the memory operation is complete.

The inquiring reader will observe that this is event-driven programming with the events managed automatically in hardware. The thread arbitrer of each microengine is able to suspend execution of any thread, if instructed so by an optional token, until an event occurs that signifies the completion of the operation. The instruction set of the microengines has similar instructions for accessing and the other types of internal or external memories and buffers. Such memories are the SRAM and the scratchpad memory as well as the receive and transmit buffers for packet I/O (RFIFO and TFIFO).

A common operational scenario of a network element is to store temporarily the analyzed packet into SDRAM memory while analyzing the headers or another part of the packet. This scenario is particularly suited for context swapping and the reason is explained with an example. Suppose that you implement a NIPS using the IXP1200 chip. A common task of a NIPS is to examine the nested headers of the packet that is accomplished by reading several bytes of the data in sequence. The network processor programmer can stack several read operations like the one described previously and append the *ctx_swap* token only in the last instruction. The thread arbitrer of the microengine will wake up the thread and resume execution only after the last memory transfer is complete. Meanwhile, while the thread is waiting for the memory accesses to complete, the other threads of the microengine can take control and perform useful work.

Similarly to the *ctx_swap* token, there are other tokens that provide additional efficiencies. For instance, the SDRAM instruction has an *optimize_mem* token that instructs the microengine to append these requests not in the default queue for the SDRAM memory accesses but to a pair of queues that distinguish between even and odd memory banks. Without deeping into details, this feature saves a few clock cycles and is near transparent to the programmer (other than appending the *optimize_mem* token).

8.1.4 Hash Unit

The *IXP1200* chip intergrates circuitry that implements a polynomial hash engine. The microengine instruction set contains instructions for hashing one to three values at once, and the values can be 48-

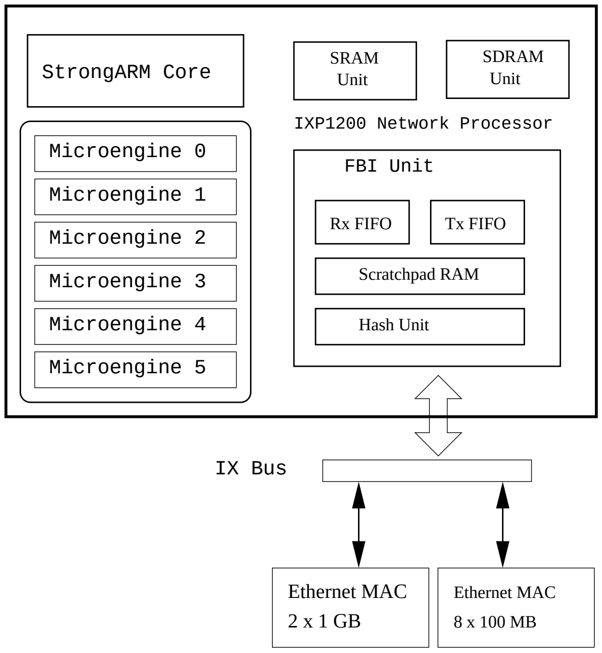


FIGURE 8.3: Typical packet data fbw in the *IXP1200*.

or 64-bits long. A microengine initiates a hash operation by writing a contiguous set of SRAM transfer registers with the data to be used to generate the hash index and then executing the hash instruction.

8.1.5 Internal Buses

The connection of the functional units of the *IXP1200* is made by high-speed, non-blocking internal buses. Buses interconnect the microengines, the scratchpad memory, the hash engine, and other internal structures. In many cases there are seperate read and write buses allowing simultaneous transfers across them. Furthermore, several functional units of the *IXP1200* have their own DMA engines, that are transparent to the programmer, but serve as hardware DMA controlers to move blocks of data between the functional units. The combination of the multiple independent buses and DMA engines ensure that external memory, not internal bus bandwidth, is the bottleneck for data throughput.

8.2 Typical Packet Data Flow in the *IXP1200*

Here, we give a simplified description of how a packet would be processed in a ’typical’ application. Figure 8.3 shows the functional units of the *IXP1200*. The following steps covers the packet data flow through the hardware:

1. The Media Access Control (MAC) device receives the data. A MAC is a piece of hardware external to the *IXP1200* that converts some physical media type to chunks of 64 bytes, called mpackets.

2. The ready bus sequencer, which is a programmable state machine on the *IXP1200*, periodically captures the ”ready flags” (MAC FIFO status indication) from each of the MAC devices. When one MAC device has at least one mpacket’s worth of data, the ready bus sequencer updates the receive ready flags. These flags serve to indicate to the microengines that one of the ports requires attention.

3. A microengine instructs the IX bus unit to transfer the mpacket from the MAC with the receive ready flag set, and store it in the receive FIFO (RFIFO). The IX bus unit takes care of the transfer and once the transfer is complete the microengine is notified.

4. The microengine instructs the SDRAM unit to move the packet from the RFIFO to SDRAM.

5. This procedure is repeated until all mpackets of the actual packet are processed.

6. After the packet has been reassembled into SDRAM memory, a microengine transfers a portion of the packet (usually the first few bytes) into its transfer registers. Then, some sort of classification is accomplished and the packet is probably modified. Next, if necessary, the microengine writes the modified bytes of the packet to SDRAM.

7. Finally the packet must be transmitted out of the *IXP1200*. A microengine checks the transmit ready flags of the destination MAC device. When the destination device is able to receive data from the *IXP1200*, the microengine requests from SDRAM to transfer a chunk of data from SDRAM to the transmit FIFO (TFIFO). When the transfer is complete, the SDRAM unit notifies the microengine.

8. The microengine instructs the IX bus unit to move the data from the TFIFO to the appropriate MAC device. Like the receive process, the transmission process must repeat itself for each mpacket in the outgoing packet.

```
.local addr $sram_data $$sdram_data
       immed[addr, SRAM_ADDR]
       sram[read, $sram_data, addr, 0, 1], ctx_swap
       alu[$$sdram_data, $sram_data, -, 10]
       immed[addr, SDRAM_ADDR]
       sdram[write, $$sdram_data, 0, 1], ctx_swap
.endlocal
```

FIGURE 8.4: Microcode Example.

8.3 Programming Languages

There are two possible ways of programming the *IXP1200*, microcode and microengine C [17]. Microcode is analogous to the assembly language of a general-purpose processor, and microengine C is analogous to a third generation language like C.

By definition, assembly language is designed to match the underlying hardware – each assembly language statement corresponds directly to a machine instruction. Unfortunately, because microengines are low-level devices, the instruction set is designed to be convenient for hardware designers, not to be convenient for programmers. As a consequence, programming in assembly language can be tedious, and is prone to errors.

On the contrary, microengine C is very similar to the classic C language. It offers type safety, pointers to memory, and functions but it is less efficient than assembly. Because we wanted to implement our system as efficiently as possible we decided to follow the ”hard-way” – assembly language.

To illustrate some of the differences in complexity between programming in microcode and microengine C, we present two versions of the same program. Figure 8.4 presents the microcode version while Figure 8.5 presents the corresponding microengine C version. In this example, we read the contents of SRAM address *SRAM_ADDR* , we substract 10 from the value, and then we store this new value on SDRAM address *SDRAM_ADDR*.

8.4 Execution and Development Environment

Digenis executes in either *simulation* mode, under the control of the *transactor*, or in *hardware* mode, on the *IXP1200* Ethernet Evaluation Board (*IXP1200EEB*). In *simulation* mode, the Developer Work-

```
    __declspec(sram) long *sram_data = (long *)SRAM_ADDR;
    __declspec(sdram) long long *sdram_data = (long long *)SDRAM_ADDR;
    *sdram_data = (long long)(*sram_data - 10);
```

FIGURE 8.5: Microengine C Example.

bench is used as the graphical user interface to control development, execution, debugging, and statistics gathering. On the contrary, in *hardware* mode, the VERA development and runtime environment is used [19, 18].

8.4.1 Simulation Mode

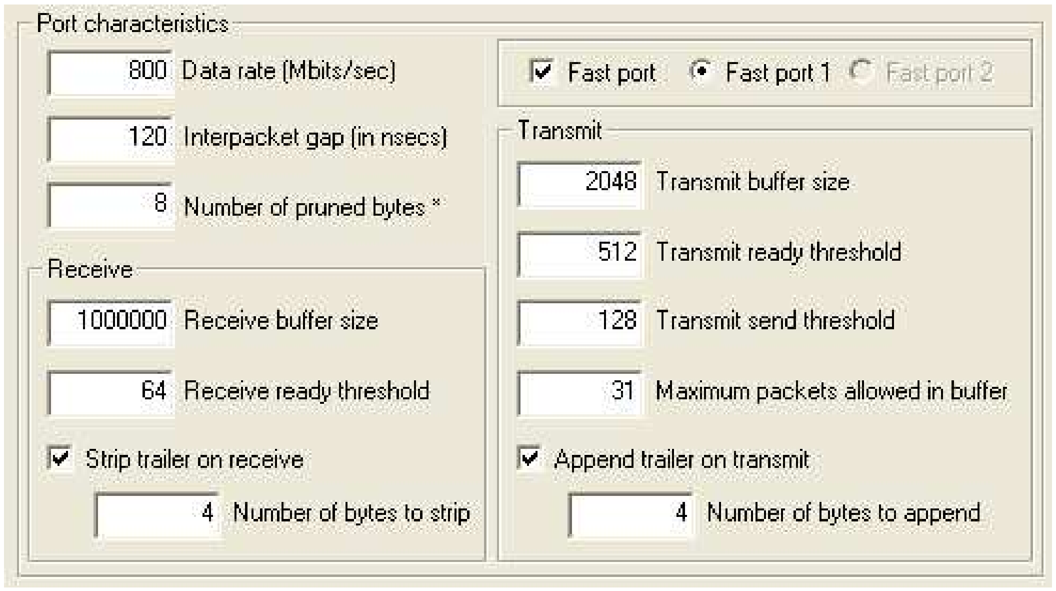
A way to evaluate *Digenis* is with the use of the *transactor*. The *transactor* is a cycle and data accurate software model of the *IXP1200* microengines, memory, buses and peripherals. The *transactor* runs under control of the Developer Workbench which also provides an IX Bus device simulation tool that generates network traffic streams onto the IX Bus interface of the *transactor*.

We used the Intel-supplied development environment on a PC running the Windows OS. This is the first step towards developing an application to run on the *IXP1200*. First, you verify that your application does not have obvious flaws using the Developer Workbench and after that, you take the next step that is to try to run the application on real hardware.

Figures 8.6 and 8.7 show the configuration of the Gigabit Ethernet and Fast Ethernet ports of the *IXP1200EEB* used during debugging and evaluation of the *DigenisNP*. As you observe, we have set a huge value to the receive buffer size. We have done this, because, due to a bug in the simulator, if this buffer overflows, the simulation fails.

8.4.2 Hardware Mode

As mentioned before, in this mode *Digenis* executes on the the *IXP1200EEB*. This board consists of an Intel IXF440 Multiport 10/100 Mbps Ethernet Controller, an Intel IXF1002 Dual Port Gigabit Ethernet Controller, 32 Mbytes of SDRAM, 2 Mbytes of SRAM and a serial port. As shipped from the factory, the *IXP1200EEB* is configured to operate as a stand-alone system in a supplied passive PCI backplane connected to a simple 100 Mbps Ethernet line card. Due to bugs contained in the software provided by Intel we decided to use an open source software called VERA.



Port characteristics

800	Data rate (Mbits/sec)
120	Interpacket gap (in nsecs)
8	Number of pruned bytes *

[✓] Fast port [● Fast port 1] ○ Fast port 2

Receive

1000000	Receive buffer size
64	Receive ready threshold

[✓] Strip trailer on receive

4	Number of bytes to strip

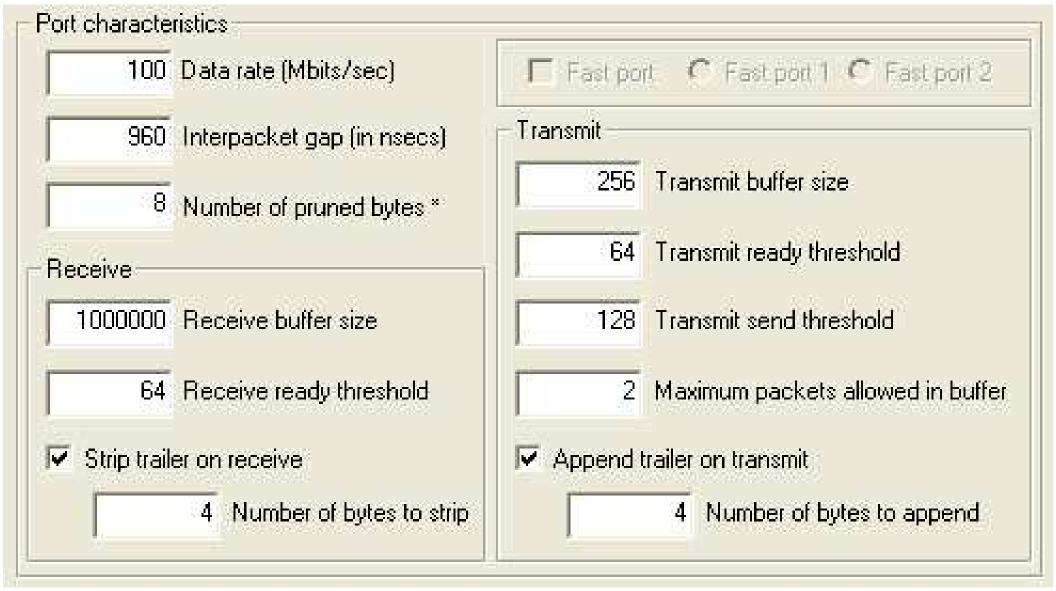
Transmit

2048	Transmit buffer size
512	Transmit ready threshold
128	Transmit send threshold
31	Maximum packets allowed in buffer

[✓] Append trailer on transmit

4	Number of bytes to append

FIGURE 8.6: Confi guration of Gigabit Ethernet Port.



Port characteristics

100	Data rate (Mbits/sec)
960	Interpacket gap (in nsecs)
8	Number of pruned bytes *

☐ Fast port ○ Fast port 1 ○ Fast port 2

Receive

1000000	Receive buffer size
64	Receive ready threshold

[✓] Strip trailer on receive

4	Number of bytes to strip

Transmit

256	Transmit buffer size
64	Transmit ready threshold
128	Transmit send threshold
2	Maximum packets allowed in buffer

[✓] Append trailer on transmit

4	Number of bytes to append

FIGURE 8.7: Confi guration of Fast Ethernet Port.

VERA requires an active PCI backplane, so, we had to reconfigure the board. Specifically, we changed the board jumpers so that it neither generated the global PCI reset signal at power-up nor acted as the PCI arbitrer – both of these functions are performed by motherboards. We also used the firmware supplied by VERA (and programmed it into the on-board Flash EEPROM) which assumes that the board is installed

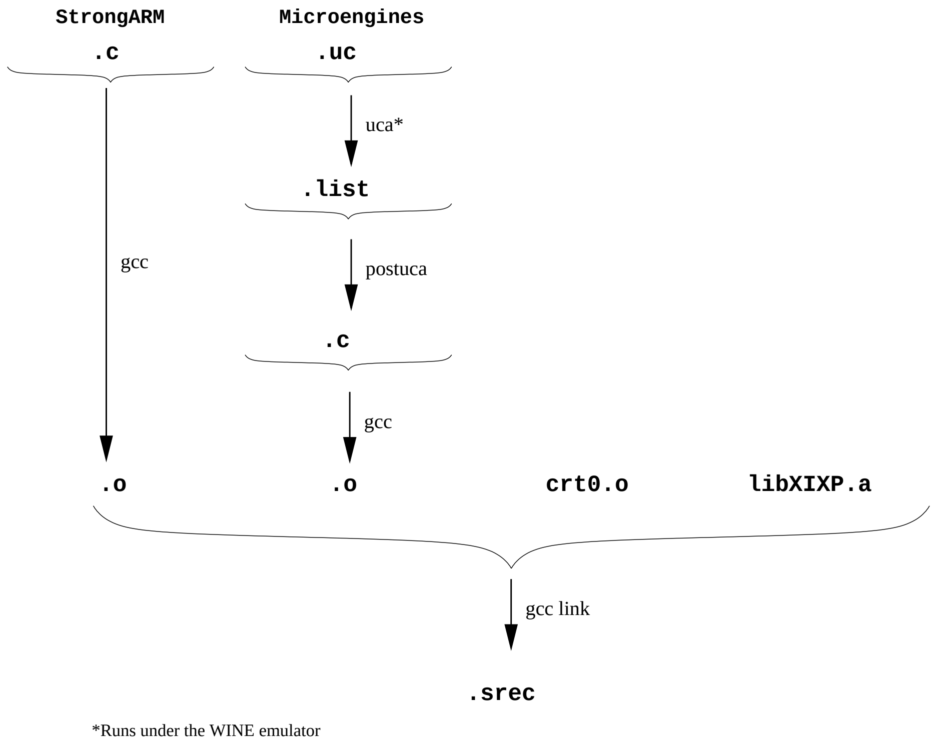


FIGURE 8.8: VERA's *IXP1200* Toolchain.

on a PC – on power-up it requests a region of the PCI bus address space from the BIOS, opens a window from the PCI bus to the SDRAM so that the Pentium can download code directly onto the board. The firmware jumps to an entry point in this downloaded code when it receives a signal from the Pentium.

For our prototype, we used the Linux OS for the development environment. We used the GNU C tools for the Pentium and StrongARM processors. The Pentium compiler is native, while the StrongARM compiler is a cross-compiler supplied by the VERA project. We used the Intel-supplied microcode assembler to assemble the *IXP1200* microcode. We ran the microcode assembler on the WINE Windows emulation environment on Linux. The toolchain to create an executable image for the *IXP1200* is shown in Figure 8.8.

We have used a device driver for the *IXP1200EEB* implemented in the VERA project in the form of a Linux kernel module. The driver exports a /dev/vera character device. Through its *ioctl* interface, this device is used to download code from the user space to the *IXP1200EEB* and give access to the memory and registers of the board during debugging. Figure 8.9 illustrates how the *sgo* tool loads and runs an executable image on the *IXP1200*.

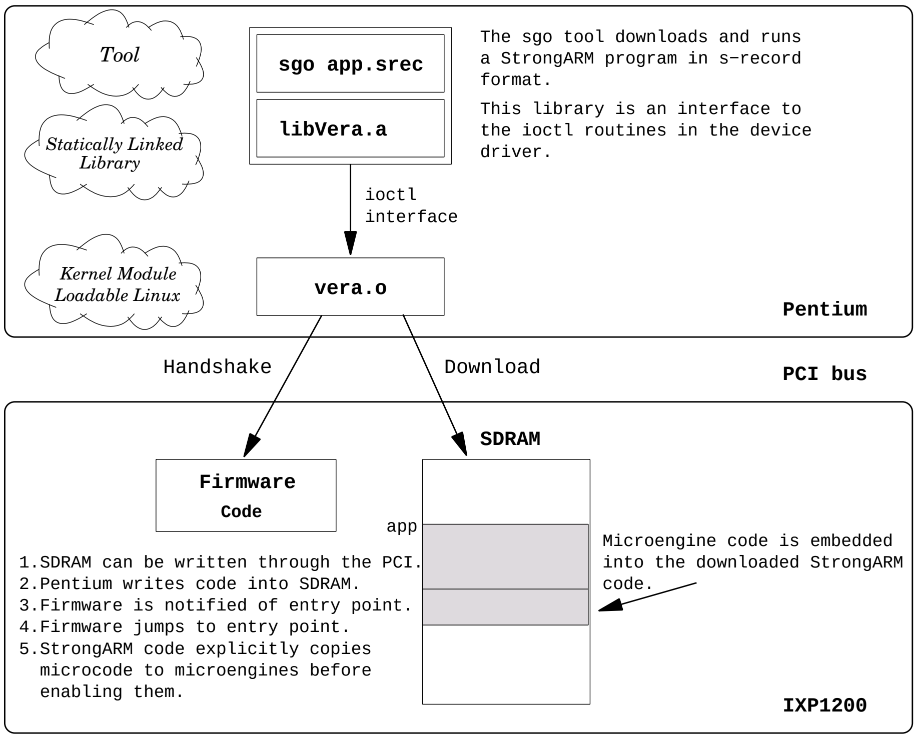

Tool
Statically Linked Library
Kernel Module Loadable Linux
sgo app.srec
libVera.a
The sgo tool downloads and runs a StrongARM program in s−record format.
This library is an interface to the ioctl routines in the device driver.
ioctl interface
vera.o
Pentium
Handshake
Download
PCI bus
SDRAM
Firmware Code
app
Microengine code is embedded into the downloaded StrongARM code.
1.SDRAM can be written through the PCI.
2.Pentium writes code into SDRAM.
3.Firmware is notified of entry point.
4.Firmware jumps to entry point.
5.StrongARM code explicitly copies microcode to microengines before enabling them.
IXP1200

FIGURE 8.9: Running an Executable Image on the *IXP1200EEB*.

9
Appendix B

9.1 Ethernet Data Rates

Because there is a general misunderstaing concerning the packet (frame) and bit rates that Ethernet supports, we believe it is a good idea to list them here. So, for example the expression ”*IXP1200* is capable of receiving traffic at 1 Gbps” is translated into ”*IXP1200* is capable of receiving traffic at 760 Mbps”.

	100 Mbps Ethernet	1 Gbps Ethernet
Bit Time	10 ns	1 ns
Byte Time	80 ns	8 ns
Interframe Gap	960 ns	96 ns

TABLE 9.1: Timing for Ethernet Packets.

	100 Mbps Ethernet	1 Gbps Ethernet
64-bytes Packets	148.8 Kpps	1.48 Mpps
1518-bytes Packets	8127 pps	81.2 Kpps

TABLE 9.2: Maximum Packet Rates for Ethernet.

	100 Mbps Ethernet	1 Gbps Ethernet
64-bytes Packets	76.1 Mbps	760 Mbps
1518-bytes Packets	98.6 Mbps	986 Mbps

TABLE 9.3: Maximum Bit Rates for Ethernet.

Bibliography

[1] Rfc 3174 - us secure hash algorithm 1 (sha1), September 2001.

[2] Aaron Turner and Matt Bing. tcpreplay Tool. `http://tcpreplay.sourceforge.net`.

[3] J. Bellardo and S. Savage. Measuring packet reordering, 2002.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[5] A. Broder. Some applications of rabin's fi ngerprinting method, 1993.

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

[7] Y. Charitakis, K. G. Anagnostakis, and E. Markatos. An active splitter architecture for intrusion detection (short paper). In *Proceedings of the Tenth IEEE/ACM Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS 2003)*, October 2003.

[8] Cisco Catalyst 6500 Series IDS Module (IDSM-2). `http://www.cisco.com`.

[9] C. Clark, W. Lee, D. Schimmel, D. Contis, M. Kone, and A. Thomas. A hardware platform for network intrusion detection and prevention. In *Proceedings of the 3rd Workshop on Network Processors and Applications (NP3)*, February 2004.

[10] D. Comer. *Network Systems Design Using Network Processors*. Prentice Hall International, 2002.

[11] Consystant Design Technologies. `http://www.consystant.com`.

[12] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel bloom fi lters, 2003.

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

[14] Intel Corporation. Intel PRO/1000 MT Dual Port Server Adapter. `http://www.intel.com`.

[15] Internet Security Systems Inc. `http://www.iss.net`.

[16] Intrusion Prevention Systems Group Test - Edition 1, NSS Group Ltd. `http://www.nss.co.uk/acatalog/`.

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

[18] S. Karlin and L. Peterson. VERA: An Extensible Router Architecture. In *Proceedings of the 4th International Conference on Open Architectures and Network Programming (OPENARCH)*, pages 3–14, April 2001.

[19] S. C. Karlin. *Embedded Computational Elements in Extensible Routers*. PhD thesis, Princeton University, Princeton, New Jersey, January 2003.

[20] R. Kemmerer and G. Vigna. Intrusion Detection: A Brief History and Overview. *IEEE Computer*, pages 27–30, April 2002. Special publication on Security and Privacy.

[21] L. Kencl. *Load Sharing for Multiprocessor Network Nodes*. PhD thesis, Swiss Federal Institute of Technology (EPFL), January 2003.

[22] L. Kencl and J. Y. L. Boudec. Adaptive load sharing for network processors. In *Proceedings of IEEE Infocom*, June 2002.

[23] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful intrusion detection for high-speed networks. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 285–294, May 2002.

[24] S. Li, J. Torresen, and O. Soraasen. Exploiting reconfi gurable hardware for network security. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2003)*, April 2003.

[25] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das. The 1999 DARPA off-line intrusion detection evaluation. *Computer Networks*, 34(4):579–595, October 2000.

[26] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Symposium on Operating Systems Principles*, pages 174–187, 2001.

[27] M. Necker, D. Contis, and D. Schimmel. TCP-stream reassembly and state tracking in hardware. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2002)*, April 2002.

[28] NetScreen Technologies. `http://www.netscreen.com`.

[29] Network Associates, Inc. `http://www.networkassociates.com`.

[30] K. K. Niraj Shah. Network processors: Origin of species. In *Proceedings of ISCIS XVII, The Seventeenth International Symposium on Computer and Information Sciences*, October 2002.

[31] V. Paxson. End-to-end Internet packet dynamics. In *Proceedings of the ACM SIGCOMM '97 conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, volume 27,4 of *Computer Communication Review*, pages 139–154, Cannes, France, September 1997. ACM Press.

[32] T. H. Ptacek and T. N. Newsham. Insertion,evasion and denial of service: Eluding network intrusion detection. Security Networks,Inc. http://secinf.net/info/ids/idspaper/idspaper.html.

[33] Radisys. `http://www.radisys.com`.

[34] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proc. of the second USENIX Symposium on Internet Technologies and Systems*, November 1999. (software available from `http://www.snort.org`).

[35] R. Russo, L. Kencl, B. Metzler, and P. Droz. Scalable and adaptive load balancing on IBM Power NP. Technical report, Research Report – IBM Zurich, August 2002.

[36] J. Santos and D. Wetherall. Increasing effective link bandwidth by suppressing replicated data. pages 213–224. `citeseer.ist.psu.edu/article/santos98increasing.html`.

[37] M. Schiffman. The million packet march. `http://www.packetfactory.net/Projects/Libnet/`.

[38] D. V. Schuehler, J. Moscola, and J. W. Lockwood. Architecture for a hardware-based, TCP/IP content-processing system. *IEEE Micro*, 24(1):62–69, 2004.

[39] C. Shannon, D. Moore, and k claffy. Characteristics of fragmented ip traffi c on internet links. In *Proceedings of the First ACM SIGCOMM Workshop on Internet Measurement*, pages 83–97. ACM Press, 2001.

[40] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a Robust Software-Based Router Using Network Processors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 216–229, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.

[41] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffi c. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 87–95. ACM Press, 2000.

[42] Time-Stamp Counter. `http://www.intel.com/design/Xeon/applnots/24161825.pdf`.

[43] TippingPoint Technolgies Inc. `http://www.tippingpoint.com`.

[44] Top Layer Networks. `http://www.toplayer.com`.

[45] Webster's Online Dictionary. `http://www.websters-online-dictionary.org`.