Analysis and Architectural Support for Parallel Stateful Packet Processing

A Dissertation Presented
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy
Doctor per la Universitat Politécnica de Catalunya

by
Javier Verdú Mulà

July 2008

Analysis and Architectural Support for Parallel Stateful Packet Processing

Author: Javier Verdú Mulà

Advisors: Mario Nemirovsky
Mateo Valero Cortés

A Dissertation Presented
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy
Doctor per la Universitat Politécnica de Catalunya

Departament d'Arquitectura de Computadors
Universitat Politécnica de Catalunya
July 2008

A l'Ester, Pares, Germà i resta de la Família
que m'han recolçat i suportat. Gràcies!

Acknowledgements

Per començar voldria agrair aquesta tesis als meus Pares, al Josep, a l'Anna i a la resta de la família per ajudar–me a créixer com a persona i pel seu recolzament en els estudis. I des de fa pocs anys a la meva nova família, pares i germans de l'Ester. També al Martín i a la Lidia, dos molt bons amics tant llunyans com propers. A tots ells GRÀCIES per haver estat amb mi en diferents moments de la meva vida.

Però sobretot a l'Ester, per haver sacrificat tots aquells caps de setmanes, vacances, ponts ... i per aquelles nits i matinades en qu havia de treballar. Així que, prepara't, ara començarem a viure!!!!

Gràcies als meus directors de Tesis, Mario i Mateo, els quals sense ells no hagus aprs a espavilar–me i per tant, a treure endavant la tesis. També a nivell professional, agraeixo a Jorge García per la gran ajuda que em va donar especialment al principi de la tesis i a Enrique per haver–me obert les portes a un viatge doctoral que avui acaba.

El començament del doctorat va suposar molts canvis en la meva vida, que van ser més fàcils de pair al conviure amb bons amics. Fran, amb qui vaig començar l'aventura del doctorat, i Germán i Oliver, amb aquelles converses nocturnes inacabables.

Des de fa temps a l'Alex i al Rubén fan que els esmorzars es converteixin en un món ple d'emocionants projectes, dels quals no descartem que algun dia es facin realitat.

També voldria fer menció d'altres amics que he trobat al llarg d'aquesta tesis, i disculpeu–me tots aquells que ara no us recordi: Josep Maria, Tana, Carmelo, Jaume, Oliver, Ayose, Daniel, Marco, Ale, Isidro, Pepe, Enric, Fernando, Pedro (que aquesta menció valgui per tots els Pedros que conec), Roberto, Llorenç, Ramón, Beti, Jordi, Silvia, Montse, Pau, Fermín, Alex Ramírez, Juli, Esther, Xavi, Ernest, Alba, Hema, Horacio, Patry, Liz, Carmina, Nay, Rubén (també que valgui per els que conec d'America), Raimir, Lemma, a els membres de sistemes i administració que m'han ajudat quan ho

he necessitat, i als que em van ajudar quan vaig he estat a Estats Units, Mario i Laura, Enric i Encarna, i Rodolfo.

Per acabar, vull agrair a l'Alex i al Rodolfo que hagin sacrificat el seu temps en la correcció d'alguns capítols d'aquesta tesis. Els seus comentaris han estat de gran ajuda.

Abstract

This dissertation focuses on the processing of packets in network nodes. The demand of packet processing differ in significant ways from other typical processing required by office applications or scientific data crunching. The system throughput requirements lead to the exploitation of parallelism through parallel architectures. Massive multithreaded processors emerge as a good alternative to provide the required resource capabilities.

Advanced network applications keep state information of processed packets. The state can be related to packets of a given flow or packets across different flows. These applications, called stateful, arise dependencies among packets and reduce the amount of available parallelism. Moreover, current execution models of network applications don't take advantage of the massive multithreaded capabilities.

In this thesis, we address the limitations of parallelism in stateful network applications to maximize the throughput of advanced network devices. Furthermore, this dissertation comprises three complementary sets of contributions focused on: network analysis, workload characterization and architectural proposal.

The network analysis evaluates the impact of network traffic on stateful network applications. We specially study the impact of network traffic aggregation on memory hierarchy performance.

We categorize and characterize network applications according to their data management. The results point out that stateful processing presents reduced instruction level parallelism and high rate of long latency memory accesses.

Our analysis reveal that stateful applications expose a variety of levels of parallelism related to stateful data categories. Thus, we propose the MultiLayer Processing (MLP) as an execution model to exploit multiple levels of parallelism. The MLP is a thread migration based mechanism that increases the positive affinity among streams in the memory

hierarchy and alleviates the contention in critical sections of parallel stateful workloads. The main goal of MLP is to maximize the throughput of massive multithreaded architectures.

This thesis provides the first step forward to overcome the performance limitations of oncoming stateful applications. In addition, it opens new research lines towards future stateful network architectures.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The evolution of the technology in communications and processing capabilities, as well as other sociologic factors (*e.g.* social progress, global networking, working connected on remote devices), leads to the fact that both the entire Internet and a large range of individual institutions nearly double the network traffic each year since 1997 [16, 61].

”Gilder’s Law” predicts that the need for bandwidth will grow at least three times faster than computing power [25]. Regarding semiconductors, ”Moore’s Law” predicts that the number of transistors that fit on a chip will double every eighteen months [56]. Even if there are studies that suggest the near end of ”Moore’s Law” [23, 52], it is expected that ”Gilder’s Law” will still hold true for the next years. Thus, as the gap between network bandwidth and computing power widens, improved processor architectures are needed to process network traffic without limiting system throughput.

The evolution of network services is closely related to the network technology trend. Originally network nodes forwarded packets from a source to a destination in the network by executing lightweight packet processing, or even negligible workloads. As links provide more complex services, packet processing demands the execution of more computational intensive applications. Complex network applications deal with both packet header and payload (i.e. packet contents) to provide upper layer network services, such as enhanced security, system utilization policies, and video on demand management. Van Jacobson suggests the beggining of a new big step forward on the evolution of Internet towards ubiquitous computing [35].

Computer architecture researches aim to maximize the system throughput to sustain

the required network processing performance as well as other demands, such as memory and I/O bandwidth. There are different processor architectures depending on the sharing degree of hardware resources among streams (*i.e.*hardware context). Multiprocessor architecture presents several processors in which there are no hardware resource shared among processors, except the main memory and interconnection bus. Multicore architectures present multiple processing engines within a single chip that share cache levels of memory hierarchy and interconnection network. Finally, multithreaded architectures integrates multiple streams in a single processing engine sharing functional units, register file, fecth unit, and inner levels of cache hierarchy. Scalable multicore multithreaded architectures emerge as a solution to overcome the requirements of high throughput systems. Nevertheless, the efficient utilization of massively multithreaded architectures[1] depends on the application characteristics. On one hand, emerging network applications show large computational workloads with significant variations in the packet processing behavior. Then, it is important to analyze the behavior of each packet processing to optimally assign packets to threads (*i.e.* software context) for reducing any negative interaction among them. On the other hand, network applications present Packet Level Parallelism (PLP) in which several packets can be processed in parallel. As in other paradigms, dependencies among packets limit the amount of PLP. Lower network layer applications show negligible packet dependencies and PLP can be maximized. In contrast, complex upper network applications show dependencies among packets leading to reduce the amount of PLP.

Multiple proposals maximize system throughput of low layer processing by exploiting PLP in massively multithreaded architectures. In contrast, they doesn't perform well with upper layer applications due to the characteristics of the packet processing.

This thesis analyzes stateful network applications focused on the upper network layers. Statefulness is the capability for keeping information about previous packet processing. The results will be used as basis to propose a new processing approach as well as to describe the required software and hardware support. The code of the application includes information that allows the mechanism to identify multiple levels of parallelism that enhance the PLP. As a result, the proposed architecture leverages the synergy among streams for both data and instruction management. That is, enhancing the probability that streams that use the same data and instructions are combined together. In addition, this mechanism provides control thread management that reduces parallelism contention in critical sections compared to other network processing approaches.

[1]We call massively multithreaded architectures to the architectures that comprise tens to hundreds of streams distributed across multiple cores on a chip

1.1 Trends of Packet Processing in the Internet

The concept of Internet as a global network of networks was created during the seventies [45]. Since then it has experienced an astonishing growth, both in traffic volume and supported applications. The Internet design is largely built on the end–to–end principle proposed by Saltzer et al. [73]. The core of the network just routes packets in a stateless fashion, whereas the intelligence of network services is assigned to the endpoints.

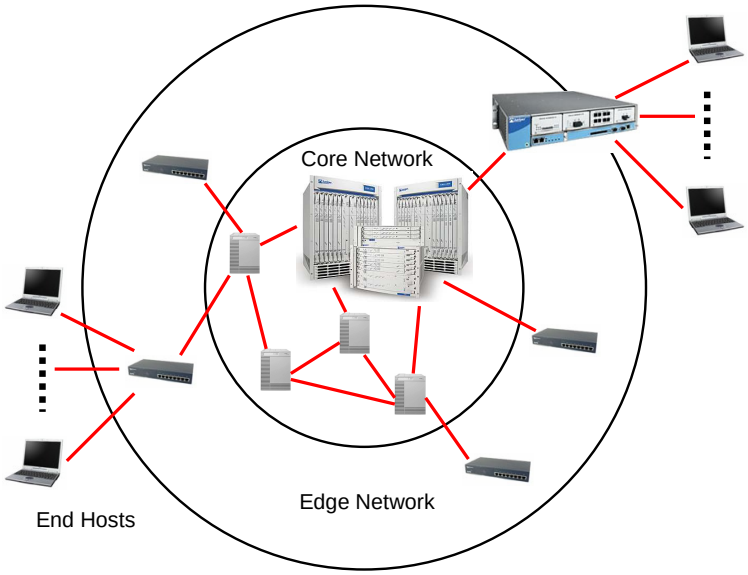


Figure 1.1: Internet Architecture Layers

Figure 1.1 shows the three main layers of current Internet design in which the previous design principle is applied. The core network comprises several powerful backbone routers that forward packets based on destination addresses, without processing the packet contents. When the packet arrives to the edge of the Internet, some additional functions are performed to the packet, such as lightweight security filtering. Finally, the end host manages the full packet contents as well as interacting with the user. In fact, the end host is the responsible of doing complex packet processing.

According to the Open System Interconnection (OSI) model (see Figure 1.2), the core of the Internet provides services limited to the layer 2–3 (*i.e.* low layer). In contrast, the end points provide services related to the upper layers of the OSI model.

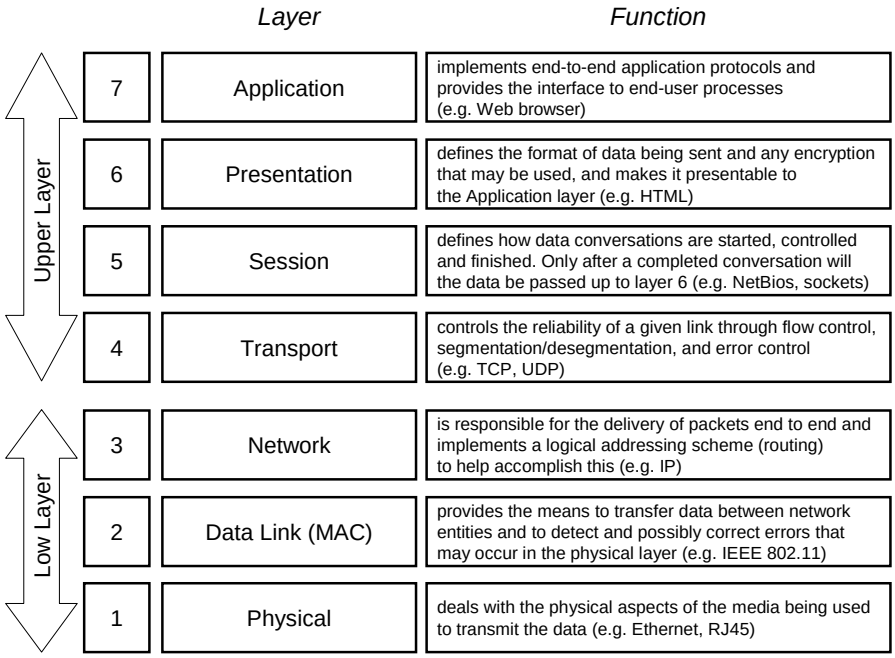


Figure 1.2: OSI Model

Due to the nowadays evolution of society and technology, the users demand new kinds of network services. For this reason, the Internet starts to experience an interesting transformation in which a part of the intelligence of the end hosts is moved to the network. Some studies show that in the near future the Internet will be like a distributed ubiquitous computing system [35]. As a result, more complex network services are assigned to the edge nodes.

Applications that provide complex network services arise two key capabilities that differ from the low layer network applications: a) deep packet inspection examines the packet payload tipically searching for a matching string or regular expression, and b) stateful processing keeps track information of previous packet processing, unlike other applications that don't keep any data about other packet processing. In most cases, deep packet inspection also integrates stateful processing. Further details can be found in Chapter 2.

To understand the impact of the Internet evolution on the packet processing requirements, we describe the following example. Consider two averaged bandwidth links of 10 Gbps (OC–192 links[2]) and 40 Gbps (OC–768 links). Assuming a traffic that presents a packet size of 500 bytes on average. Then, the network links require to sustain 2,5 Million Packets Per Second (MPPS) for OC–192 links and 10 MPPS for OC–768 links. That is, a given packet has about 400 ηsec and 100 ηsec to be processed, respectively.

On the other hand, lighweight packet processing requires tens of instructions, whereas complex applications need thousands of instructions [53]. Besides the above mentioned packet processing time requirements, we assume a packet processing workload of 10K instructions, and a processor frequency is 1GHz. As a result, the required processing performance is about 25 Instructions Per Cycle (IPC) for OC-192 links and nearly 100 IPC for OC-768 links. Therefore, processor architectures must provide high system performance to maximize the sustained throughput.

1.2 Emerging of Massive Multithreaded Architectures

Network processing requires high throughput as we discussed in the previous section. The computer architects aim to maximize the throughput of systems by properly exploiting parallel processing paradigms. However, a variety of dependency sources reduce the amount of parallelism that can be exploited.

There is a wide range of applications that exhibit a large Instruction Level Parallelism (ILP). Superscalar processors address this fine–grain parallelism by executing small sections of code of a given thread[3] in parallel. The compiler is able to significantly reduce the impact of control dependencies (i.e. the execution flow is changed by a control flow instruction, such as conditional branch) and data dependencies (i.e. an instruction can start its execution when all its input dependencies are resolved) by rearranging the instructions. In addition, superscalar processors include mechanisms for speculatively execution of instructions to further exploit ILP (*e.g.* branch predictor). Unfortunately, the available ILP from a thread is limited to the mentioned dependencies, and other factors such as hardware area and complexity constraints [96].

Other workloads, such as scientific applications, show large Data Level Parallelism

[2]Optical Carrier (OC) levels describe the signal rate multiples for transmitting digital signals on the network. The base rate (OC–1) is 51.84. Other levels are given as OC–n.

[3]Throughout this thesis we use the word thread to refer to the software context and stream to refer to the hardware context

(DLP) since a given computation is repeated on multiple data items of large data structures, such as vectors or matrices. Vector processors address this paradigm with hardware and compiler support that properly uses an specialized ISA (*e.g.* Multimedia Extensions, such as MMX [32], 3DNow! [4] by AMD, and SSE and SSE2 [33] by Intel Corp). Vector processors are capable of hiding the memory latency by maximizing the use of available memory bandwidth as well as reducing branch mispredictions by using vector mask registers and predicated execution instead of conditional branches.

Even if the performance can be increased with ILP and DLP, there are systems that need to exploit further levels of parallelism than the ones comprised within a sigle thread. Multithreaded architectures are a good alternative, since they pursue increasing the system performance by executing instructions from more than one thread, also known as Thread Level Parallelism (TLP). TLP is orthogonal to the parallel paradigms exploited inside a given thread (*e.g.* ILP, DLP). Depending on the architectural approach, multithreading can either increase the performance of a single program thread by concurrently executing several threads from a single sequencial program (*e.g.* trace processor [71], speculative multithreaded processor [49]) or increase the performance of a multiprogramming or multithreaded workload (*e.g.* HEP system [80], Sun MAJC [88], Piranha [9]). Multicore and/or multithreaded architectures are suitable for this type of applications.

Recently there are processors that include additional resource sharing levels between multithreaded and multicore processors (*e.g.* Sun UltraSPARC T2 [26]), where the streams are partitioned into two pipelines and groups. The multithreaded processors in which threads share the issue width present different execution approaches. A coarse–grain multithreaded processor [2, 85] issues instructions from a single thread in a given cycle. The processor switches to a different thread when it experiences a long latency operation (*e.g.* an outer cache miss). In contrast, a fine–grain multithreaded processor [3, 66, 42, 80] switches to a different thread after each instruction fetch. Finally, a simultaneous multithreaded processor [78, 104, 103, 29, 90, 22] can issue instructions from multiple threads in a given cycle.

Regarding the network workloads, the architectures exploit the inherent parallelism of network processing (i.e. PLP) through the execution of parallel applications. Early packet processing relied on special–purpose architectures, since network processing presented lightweigth computational workloads, that is reduced number of instructions with similar behavior among packet processing. In order to provide more flexibility and lower implementation cost, programmable, high-throughput Network Processors (NPs) with

specific network capabilities [21] were developed such as the Intel IXP network processor family [34] and the IBM PowerNP family [31]. The NPs exploit the TLP of network workloads by leveraging multicore and/or multithreaded architectures.

NPs mostly focus on low network layer packet processing. As new network services become common, NPs are not capable to provide the required capabilites for intensive upper layer network workloads. For example, NPs are optimized to execute reduced workloads that present well differentiated pipeline stages. In contrast, upper layer processing presents large complex workloads with high demand for memory access. Thus, architectures capable of running large number of streams in parallel are needed to sustain the packet processing throughput required by the evolution of the Internet as mentioned in the previous section. For this reason, massively MT architectures provide the best alternative to contemporary network workloads.

Massive multithreaded processors comprise tens to hundreds of threads distributed across multiple cores on a chip. In fact, there are several massive multithreaded processors mostly focused on network processing, such as Consentry Networks processor [18], UltraSparc T1&T2 [86, 26], Cavium [12], Tilera64 [8]. One of the main challenges for these processors is to maximize the utilization of large amount of available resources by properly distributing parallel network workloads across multiple cores.

1.3 Problem Statement

While network applications can take advantage of packet level parallelism, stateful workloads present limitations that reduce the system throughput. Not only packet dependencies limit the amount of parallelism, but the significant variations on packet processing leads to significant negative interaction among threads. That is, the processing of several packets can demand quite different instructions and data working set. For example, the processing of a given packet may need to update statistics of a particular user, whereas the following packet may only need to update the statistics of the connection and trigger an action.

The natural question is: *how can we exploit massively multithreaded architectures to maximize system throughput of stateful packet processing?*

This dissertation answers to that question focusing for the first time on stateful applications. The thesis analyzes their behavior and processing requirements and then proposes a suitable software and hardware support to improve their performance running

in massively multithreaded architectures. The dissertation demonstrates that stateful applications integrate multiple levels of parallelism suitable to be exploited in the above mentioned architectures.

1.4 Contributions

This is the first thesis that challenges maximizing system throughput of stateful applications on parallel architectures. This dissertation describes the work done to achieve this goal which comprises three main steps: firstly, analyzing network traffic properties that affect stateful packet processing; secondly, workload characterization of network applications and especially of stateful deep acket inspection workloads; thirdly, the description of the MultiLayer Processing (MLP) proposal.

1.4.1 Primary Contributions

This thesis makes three main contributions:

1. **Network Traffic Analysis.** Acquiring network traffic traces has always been a difficulty to overcome, due to technical reasons and confidentiality issues. Several network sites, such as NLANR [59], CAIDA [11], and RedIRIS [68] provide publicly available traffic traces with different network bandwidth. The traces comprise just a few bytes of each packet with anonymized IP addresses, thus preventing the study of spatial IP address distribution. As a result, experimental packet processing based on IP addresses can show misleading results.

   The behavior of stateful DPI workload is related to the network connections, unlike the lower layer applications that are sensitive to IP address distribution among packets. Thus, it is important for stateful research to analyze the impact of traffic aggregation (i.e. number of active flows within a window of packets) on the network processing performance.

   We present the first evaluation of the mentioned network properties of stateful applications. We evaluate the impact of sanitized traffic processing (i.e. anonymization of any private data, mainly IP addresses of the packets). The goal is to validate such traffic traces for being used in research proposals of stateful applications.

Then, we study the impact of traffic aggregation on a variety of network application categories. We analyze the differences between stateful processing and other network applications. Further details can be found in Chapter 3.

2. **Workload Characterization.** There are several network benchmark suites in the community, such as CommBench [100], Netbench [54], and NpBench [44]. The authors characterize each benchmark and analyze architectural bottlenecks. However, the mentioned suites do not include stateful applications and the authors do not study the differences among applications according to the network layer processing.

   Throughout Chapter 4 we propose a classification for better identifying the requirements of network applications. This is the first taxonomy proposal of network workloads according to data management during the packet processing. We show there are several categories of network processing workloads depending on the data management. We analyze representative network workloads according to the proposed classification and, for the first time in the research community, stateful DPI applications are characterized.

   Stateful DPI presents different workload behavior depending on the goal of the application. A number of stateful applications show constant workload when most of the traffic presents similar packet processing, such as network monitoring. Yet other stateful applications can show significant workload variations (*e.g.* network security), due to the knowledge obtained by previous processing of packets of the same flow. For example, complex network security presents computational intensive workloads. However, once a given flow is marked as either a safe connection or an attack hazard, it is likely to execute lightweight workloads.

   In addition to the previous mentioned study, we characterize different stateful DPI applications throughout the life of a particular network connection. We will especially discuss issues related to the impact of statefulness on the memory hierarchy. The conclusions provide key understanding of the processing requirements of non–constant stateful workloads. Further details are available in Chapter 4.

3. **MultiLayer Processing Proposal.** Several processing approaches aim at maximizing parallel packet processing. They exploit the PLP by taking advantage of well differentiated pipeline processing stages in lower network layer applications. As we demonstrate in this thesis, upper layer workloads significantly differ from lower layer workloads (*e.g.* more complex computational workload presents dependencies that makes pipelining hard). One of the outcomes is that proposed

mechanisms have to consider the additional characteristics of stateful applications.

Parallel stateful DPI applications expose additional dependencies that can reduce the amount of PLP. Instead, stateful DPI workloads present a different type of parallelism that can be exploited in addition to PLP. Stateful DPI applications comprise a variety of different code categories that are closely related to the TCP/IP network layer stack. Each code category is called processing layer. Although there are dependencies within the same processing layer, there are marginal dependencies between different layers. Current compilers cannot detect processing layers, since they are related to network knowledge about the stateful data structures. But, the developer can introduce information in the code to provide the processing layer identification.

We propose the MultiLayer Processing (MLP) as an execution model to take advantage of the additional levels of parallelism of stateful applications. In fact, this is the first approach focused on stateful applications that maximized the throughput of massive multithreaded architectures. The mechanism is based on thread migration and it is complementary to other proposals for network processing, such as software pipelining. We describe the software and hardware support for MLP implementation. The methodology uses the information of the code to split the packet processing workload into a variety of processing layers. Then, a thread that processes a given packet migrates from one core to another according to the processing layer of the workload. The key issues of this approach are to identify the variety of exploitable processing layers and to properly distribute the workload among threads.

The goal of MLP is to increase the system throughput by a) increasing the positive aliasing among threads in the memory hierarchy, and b) alleviating the contention of parallel workloads in critical sections. The experiments assume stateful DPI benchmarks with sub–optimal parallel design and manual processing layer detection. We will show that MLP presents better performance scalability than other proposals for parallel network processing. Further details can be found in Chapter 6.

Figure 1.3 shows how the main contributions of this thesis fit together. The network traffic analysis points out the key knowledge about the impact of traffic properties on stateful applications. The conclusions lay down the basis of using representative traffic traces to characterize network applications. The studies of stateful workloads describe the requirements and behavior of this network workload category, and present the differences against other network applications. The analysis of stateful DPI codes leads to the

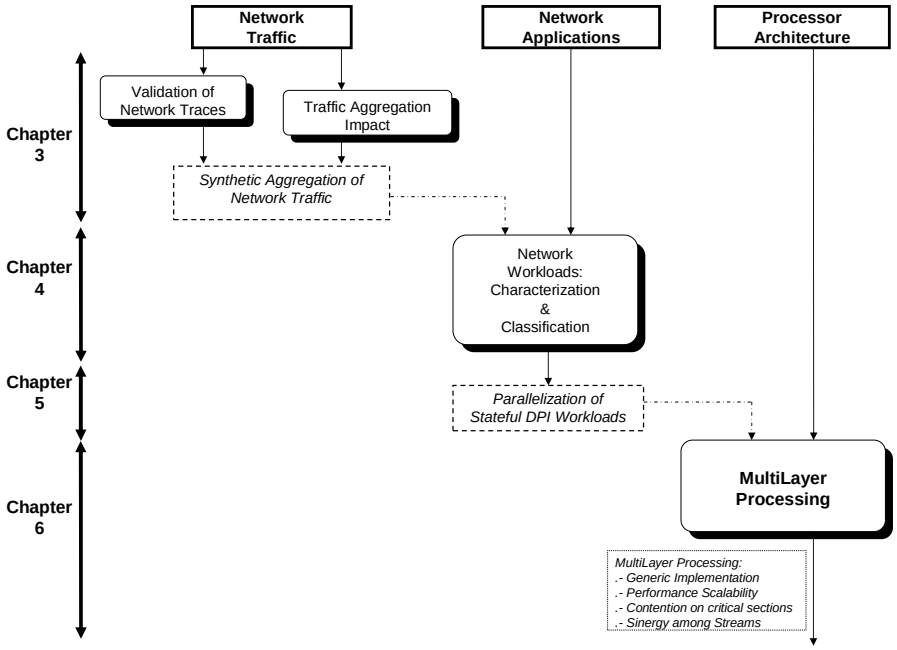


Figure 1.3: Thesis Overview

parallelization of current single–threaded stateful DPI benchmarks. Finally, we combine the knowledge of network traffic and workload characterization to propose a mechanism that maximizes the system throughput of stateful DPI in massive multithreaded architectures.

1.4.2 Other Contributions

The thesis also presents other contributions:

1. First parallel implementation of a publicly available Network Intrusion Detection System (NIDS) called Snort [10, 81] running on processors with tens to hundreds of threads. This contribution overcomes the lack of parallel stateful applications, especially because there are emerging proposals on running Snort on multithreaded architectures. It has already been used in several projects and it will be likely used in many other studies.

2. First technique that increases the traffic aggregation of a network link and overcomes the lack of representative publicly available traffic traces. The mechanism linearly scales the bandwidth of the base traffic. The resulting aggregated traffic presents larger amount of unique flows between two packets of the same flow.

3. First taxonomy proposal of network workloads according to data management during the packet processing. This classification allows the identification and characterization of the different packet processing stages within the stateful DPI applications.

1.5 Organization

Chapter 2 provides background information to better understand the disertation. We provide information about stateful DPI and describe limitations of packet level parallelism of such applications.

Chapter 3 analyses network traffic traces. First, we validate sanitized network traffic to be used with stateful applications for research purposes. We describe a mechanism that synthetically generates traffic traces with more active flows than the original ones. Finally, we study the impact of several traffic aggregation levels on memory hierarchy performance.

Chapter 4 characterizes network workloads. Network applications present a variety of workload requirements. Thus, we introduce a network workload categorization to classify applications according to data management. We analyze the key differences among categories in order to understand why current network processors are not suitable for stateful processing. We also characterize the application behavior throughout the lifetime of a TCP connection. The results point out significant variations on processing requirements among related packets.

Chapter 5 introduces background information of parallel stateful DPI applications. We demonstrate the need of parallel stateful DPI and describe key characteristics of the parallel code. Besides, we present current proposals that distribute parallel network applications into multithreaded architectures.

Chapter 6 presents the MultiLayer Processing (MLP). We present the idea of MLP as well as describe the software and hardware support for MLP implementation. We discuss tradeoffs of load–balancing using the MLP model in a massive multithreaded

architecture. Finally, we evaluate this mechanism and analyze the results compared to current proposals for parallel network applications.

Chapter 7 summarizes the conclusions of this disertation and outlines future topics for research. In fact, we are already working on some of them.

Chapter 2

Background

This dissertation focuses on the processing of packets in network nodes. The demands of packet processing differ in significant ways from other typical processing required by office applications, or scientific data crunching. In this chapter we set the stage by reviewing the basics of packet processing.

This chapter is organized as follows. In Section 2.1 we address a particular type of packet processing, called deep packet inspection, followed by a discussion of stateful packet processing in Section 2.2, and a specific discussion of the issues raised by dependencies between packets in Section 2.3.

2.1 Deep Packet Inspection

The formidable expansion of the Internet is largely due to its architecture, based on a few basic principles that guaranteed its resilience and its scalability. Foremost among them is the ”end–to–end” principle [73], which pushes most of the processing to the endpoints (*i.e.* computers). The end–to–end principle assigns to the intermediate nodes (*i.e.* switches, edge routers, and backbone routers) only the most basic operations, limited to the layers 2-3 of the Open System Interconnection (OSI) model, as shown in Figure 1.2. According to this model, a router as a trusted mailperson looks only at the header (envelope) in order to deliver (route) the packet. It is also a memory less (stateless) mailperson, in that it does not keep track of the fate of the packets. Thus, a router

strictly implemented according to the end–to–end principle, has no business in inspecting the content of a packet, or in keeping any sort of state. The end–to–end principle, which was a radical departure break from the centralized architecture of the existing telephone network, served well the Internet through its nascent and rapid growth periods. Pushing the responsibilities of guaranteeing a reliable transport to the endpoints allowed for a rapid expansion of the network and enabled an explosive growth of the applications.

Nowadays, the Internet touches every aspect of daily life. From the playground of mutually trusting early researchers, it has become the lifeblood of business, social interaction, and cultural and academic organizations. This expansion has brought its own malaise, in that the Internet is no longer the safe sandbox of yore. Today we expect much more than the ”best effort” delivery of a message or a file. Financial transactions, video streaming, interactive voice and video impose demands that far exceed those of the early years. Sure enough, the available bandwidth, due to the advances in optical communications, is many orders of magnitude higher than in the early days [61]. Yet, the need for differentiated services has challenged the end–to–end principle. So has the need to support security. The Internet growth in volume and applications was accompanied by a proliferation of so called ”middle-boxes”, which in many ways violate the end–to–end principle. The trend towards ubiquitous computing accentuates the push of complex services from the end points to the Internet cloud. Van Jacobson, a primary contributor to the TCP/IP stack, offers a compelling perspective of this trend [35].

To address this evolution, network applications push to process upper layer network layers involving deep packet inspection (DPI). We call DPI to those operations that examine the packet payload, typically searching for a matching string or regular expression. This is akin to our mailperson opening the envelope and actually reading the content of the letter. Pattern matching has some heavy processing requirements, but it does not necessarily require keeping state.

Lower layer applications deal with well known located fields in the packet header. For instance, a router forwards packet by searching an entry in the forwarding table according to the IP destination address. On the contrary, DPI demands much more intensive processing to deal with payload fields. DPI presents complex field recognition process due to non–well known formats of payload. That is, there is a large variety of protocols that lead to different packet payload formats. Moreover, DPI may require additional processing requirements such as keeping statistics or information about the packets.

Figure 2.1 depicts packet processing stages, namely: receiving (Rx), decoding, processing, and transmitting (Tx). Receiving and transmitting show no special requirements, since they only read/write from/to memory buffer associated to the network card. The decoding stage, although being actually part of the packet processing, it is depicted as an isolated stage since it is performed before any packet processing. As we previously mentioned, DPI tends to show heavy workload at the processing stage versus lower layer applications.

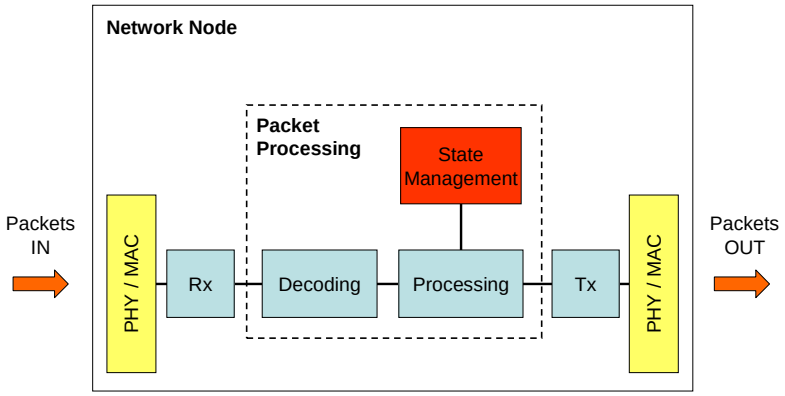


Figure 2.1: Packet Processing Flow Path

2.2 Statefulness

We define flow as the communication between two network end points, regardless the connection protocol used (*e.g.* TCP, UDP). Consider now an application that demands regular expression matching across a TCP stream, involving many packets in that TCP connection. This type of pattern matching requires keeping state across packets in the flow (intra–flow information). Other applications, particularly in the security area, may require maintaining also inter–flow information. Thus, while DPI processing goes into the payload of the packet without keeping state, stateful DPI maintains state information

of packets within a flow, or packets across different flows. In most cases, DPI also involves stateful DPI.

A stateful application requires maintaining a data structure by checking/updating information. In contrast, a stateless application accesses to data structures only to compare packet contents to find a particular pattern (*e.g.* lookup table or rule based pattern matching). The state information kept may be simple, such as the flow state required by layer 4 applications, or more complex, such as user state required by upper layer applications. In Section 4.4.1 we discuss state categories handled by stateful applications.

Security, monitoring/billing, enhanced VoIP, management policies are some examples of stateful network processing. The action triggered by the processing of a packet depends on the packet itself and previously processed packets. Upper layer network processing tends to show more statefulness in handling the relation between packets. There are many other areas where applications may perform stateful processing besides networking, such as e-commerce and databases. In general, it is likely that applications that process data input items may require stateful processing to provide enhanced services.

This thesis focuses on stateful processing performed by Network Intrusion Detection Systems (NIDSs) as representative stateful DPI applications. NIDSs trigger alerts when detects malicious activity, attacks, or anomalous traffic. NIDS is an ever–evolving area in which stateful processing is a key requirement to detect sophisticated attack methods in several security areas, such as malware, anomaly traffic detection, stateful worm detection, stateful packet inspection. The attackers take advantage of stateless firewalls that cannot look beyond the single individual packet while inspecting network traffic. For instance, in 2001 a worm called Code Red infected over 350,000 hosts over a week and the infection rate was doubling in about 37 minutes. Two years later, the Sapphire/Slammer worm infected more than 90 percent of vulnerable hosts in the world within only 10 minutes [30]. More complex NIDS with stateful capabilities are being developed to catch these most recent attacks.

2.3 Dependencies Among Packets

Lower layer applications can exploit packet parallelism since processing among packets are completely independent. However, both DPI and stateful DPI applications introduce important processing requirements that can limit parallelism among packets. For the

sake of readibility we describe state–dependency and order–of–seniority processing.

Depending in the degree of statefulness, the application maintains information of middle layers (*e.g.* flow) or upper layers (*e.g.* user). The higher the layer, the higher the probability is that a particular state is shared by two or more in–flight packets, or that two packets belong to the same user than to the same flow.

Order–of–seniority requires a particular input of data to be processed following a certain order. In the network environment, the packets sometimes have to be processed in the order of arrival to the network node. This feature specially impacts the performance of systems that exploit packet processing parallelism. Due to a variety of reasons, a given packet can be processed faster by a thread than others. That is, a thread can reach a checkpoint before other threads, which are processing ”older” packets. If the checkpoint needs order–of–seniority, the thread has to wait until the processing of previous packets reaches that checkpoint.

In general, checkpoints are associated to updates of the stateful data, and the consistency of the data is determined by the original order of arrival of the packets from the network. However, order–of–seniority is not a requirement in all stateful applications.

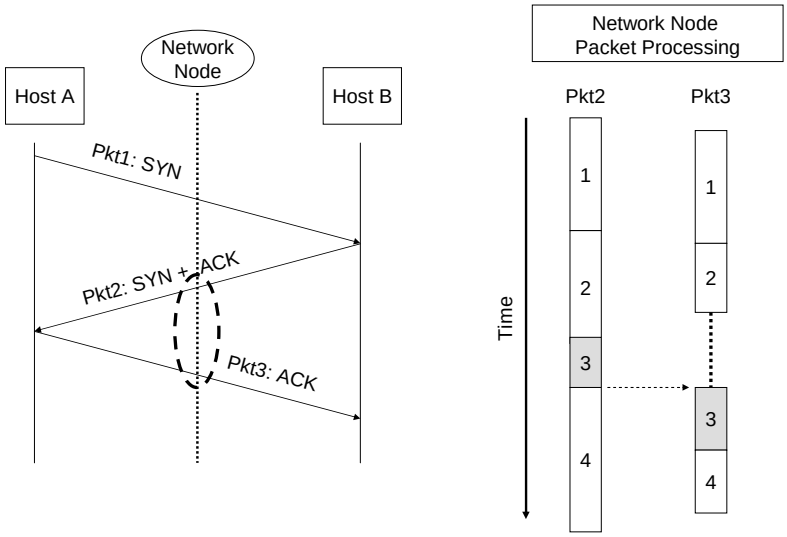


Figure 2.2: Example of Order–of–Seniority Conflict

For example, NIDSs that detect TCP protocol based attacks need order–of–seniority to update stateful data. If the three–step TCP handshake protocol is not correctly performed, an alert is triggered. Otherwise, the system can trigger false positive alerts (*i.e.* a false attack is detected) and false negative detections (*i.e.* a real attack is not detected). Figure 2.2 depicts this scenario. On the left part of the graph, we can observe the three–step TCP handshake protocol between two hosts A and B. In the middle, there is a network node that performs stateful packet processing for security services. The right part of the graph shows the packet processing of second and third packets. In this scenario we assume both packets are processed in parallel. Each numbered block represents a section of code for processing the packet. The section number 3 requires order–of–seniority processing (*e.g.* updating the flow state). For different reasons, although packet 3 starts to be processed after packet 2, it can process the section 3 earlier than the other packet. Then, this packet processing has to be stalled until the packet 2 has finished the section 3.

To recapitulate, emerging complex network services present stronger likelihood of experiencing packet depencencies that can significantly reduce the exploitation of packet level parallelism, especially in stateful DPI applications on the highest network layers.

Chapter 3

Network Traffic Analysis

Representative network traffic traces are mandatory to do research in network processing. Applications present significant performance variations according to the network traffic features. Thus, it is critical to preserve key real traffic characteristics.

For reasons of confidentiality, most of the original packet data (both header and payload) is faked or suppressed in the publicly available traffic traces. For example, IP addresses are anonymized in order to avoid revealing any real IP address. The anonymization lead to lose some key network traffic properties, such as IP address distribution. Indeed, network research community has demonstrated that anonymized traffic traces are not useful to do research in lower network layers, but there is no study about the impact on stateful.

The problem is stressed in stateful applications. Packet processing requires high traffic bandwidth to simulate traffic of current and future network edge–nodes as well as bidirectional traffic (*i.e.* source to destination traffic and vice versa) to correctly keep track of states due to statefulness. Nowadays, the sites that provide public traffic present many traces with unidirectional high bandwidth traffic and few traces with bidirectional low bandwidth traffic.

This chapter addresses the effects of the previously mentioned network features on network applications, especially stateful processing. We analyze the impact of anonymized IP addresses on the memory workload of stateful processing. We also study the impact of network traffic aggregation on the memory performance of several network applications. In order to do this study, we present a mechanism to linearly increase traffic

aggregation from a particular bandwidth link to another.

3.1 Chapter Roadmap

Section 3.2 introduces key network properties that are under study in this chapter. Sections 3.3 and 3.4 show the two studies of this chapter focused on the effects of sanitized traffic processing and the impact of traffic aggregation on network processing, respectively. The former compares the data memory workload of sanitized traffic to real traffic processing. The latter shows the impact of traffic aggregation on the memory performance of several network applications, especially comparing layer 2 to stateful layer 4+ network processing.

Related work is presented in Section 3.5 and a summary of the chapter in Section 3.6.

3.2 Network Properties Under Analysis

Network traffic features have serious implications in the analysis of network applications and systems. Network workloads are sensitive to a variety of traffic characteristics. Thus, if experimental traffic has a lack of critical features, the results can be misleading. This section introduces two network traffic properties that affect the performance of networl applications.

3.2.1 Sanitization

Sanitization is the procedure to anonymize any content with private data. In the network area, sanitization means the modification of packet contents in order to skip any private data (e.g. real IP addresses). Moreover, most of the publicly available traffic traces show few bytes of every packet (*i.e.* 40 bytes of TCP/IP header plus a parametrized number of bytes of the payload) to skip private payload. Thus, sanitized traffic presents fake source/destination IP addresses and therefore it involves the loss of real IP address distribution [39].

Leland et al. [46] studied the Ethernet traffic distribution and discovered self–similarity and in both LAN and WAN traffic. That is, no matter what time scale you

use to examine the data, you see similar patterns. Kohler et al. [39] explore multifractal models and present an example of self–similar IP address distribution (Figure 3.1). In this figure we can observe histograms of IP addresses with two succesive 32x magnifications from Figure 3.1(a) to Figure 3.1(c). There is a box for every non–empty address prefix shown in the X axis. The Y axis is the prefix length. We can see the prefix distribution patterns are similar in every prefix scale.

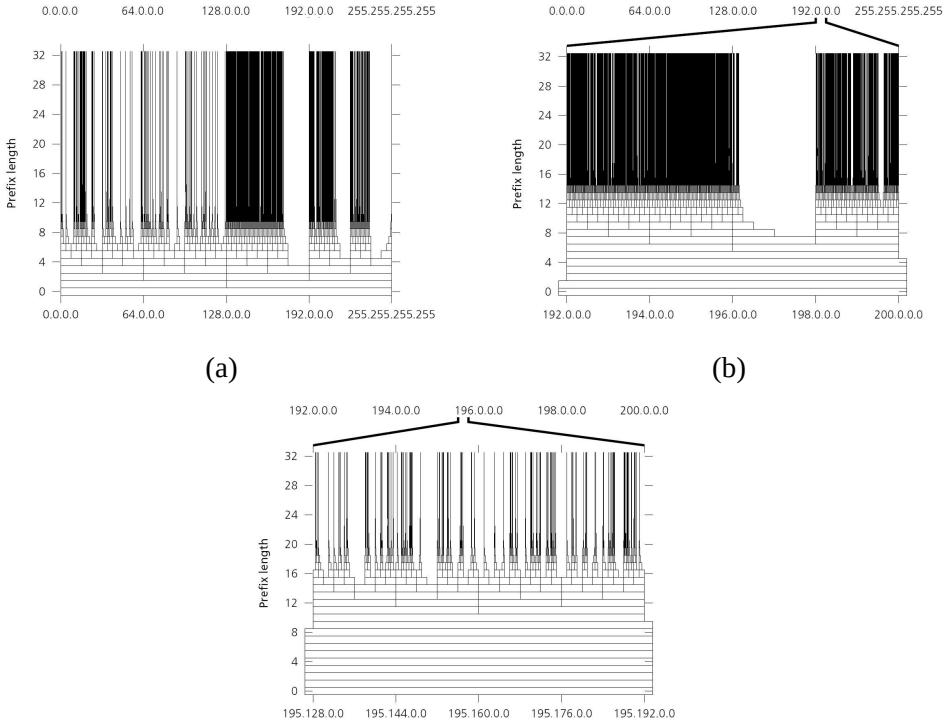


Figure 3.1: IP address structure of a traffic trace (directly taken from [39])

There are several sanitization proposals. Some of them are: K. Cho [15] anonymizes tcpdump traces by stripping packet contents and rewriting packet header fields; the Crypto-pan [102] methodology provides a consistent prefix–preserving scheme by using a shared cryptographic key; Pang et. al. [62] propose transformation scripts that operate on application–level data elements and include packet payload in the result. Overall, methodologies cannot keep all the real traffic features and some of them are lost.

Sanitized traces can be useful for some study purposes, since sanitization does not

modify temporal locality traffic properties. However, when the packet processing depends on IP address distribution (e.g. IP forwarding), sanitized traces can lead to unfair results. The question that comes out is whether the sanitized traces can be used as representative traffic for stateful processing. In Section 3.3.2 we answer to this question.

3.2.2 Traffic Aggregation

We define traffic aggregation as the distribution of packets from unique flows during a period of time. On one hand, it is related to the bandwidth link. Higher traffic bandwidth shows more packets during a time interval. On the other hand, traffic aggregation is also related to the network utilization. The more users connected and the more network utilization, the more unique flows in a period of time. Thus, traffic aggregation is defined by the unique flow rate and the packet inter–arrival distribution.

We define unique flow rate as the ratio of active flows within a window of packets (*i.e.* traffic window), as shown in Equation 3.1. The traffic window size is defined by the traffic interval between two packets of the same flow. A ratio of 0 denotes that packets of a given flow are consecutively received. A ratio of 1 means that each packet belongs to a different flow. The distribution of unique flow rates over the traffic is modeled by the packet inter–arrival distribution.

$$Unique\ Flow\ Rate = Avg\left(\frac{Unique\ flows}{Traffic\ Window\ Size}\right) \tag{3.1}$$

Figure 3.2 depicts two different scenarios of unique flow rates. On the left of the figure, there is a of packets (boxes) of different flows (letters). We assume the window of packets is defined by the packets between two packets of the flow ”A”. On the top of the figure, there is a scenario in which every packet of the window belongs to a different flow. Therefore, the unique flow is 1. On the bottom of the figure, we can observe other scenario that shows several packets that belongs to the same flow (”B” and ”C”). In this case, the unique flow is 0.5.

Packet inter–arrival distribution models how packets are scattered over the traffic. Crovella et al. [63] demonstrate that the presence of self–similarity at the link layer depends on whether reliable and flow–controlled communication is employed at the transport layer. In the absence of reliability and flow control mechanisms (e.g. UDP protocol) traffic shows less self–similarity compared to reliable communication (e.g. TCP protocol) that preserves long–range dependency as scale invariant. From this definition we

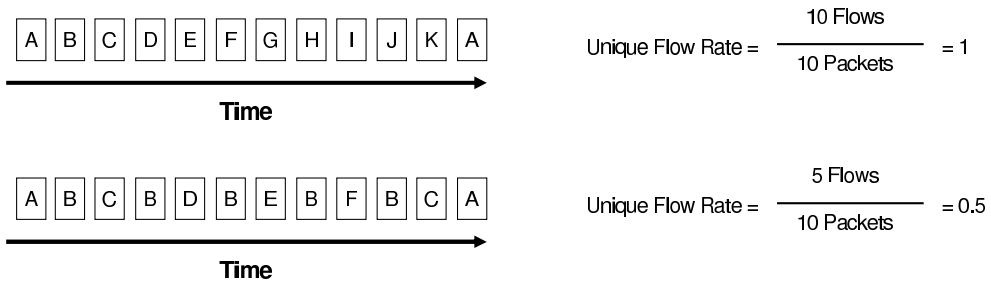


Figure 3.2: Example of Unique Flow Rates

distinguish intra– and inter–flow temporal distribution. The former shows packet distribution from a particular connection given by the flow control mechanism of the transport protocol (e.g. TCP Window). The later shows the distribution of packets from multiple flows between two packets of the same flow. More than 50% of the traffic presents distance between two packets of the same flow from hundreds to thousands of packets in low and high bandwidth links, respectively. Nevertheless, the distance between two packets of the same flow is inherently sensitive to the intra–flow temporal distribution.

Traffic aggregation increases inside of the Internet core. However, current Internet evolution leads to increments of traffic aggregation on the edge–nodes of the network. Thus, high traffic aggregation shows high unique flow rate between two packets of the same flow. We analyze the impact of traffic aggregation on memory hierarchy performance running a variety of network applications with different processing requirements.

3.3 Validation of Sanitized Traffic Traces

In this section we study the overhead in stateful workload due to sanitized traffic traces. Firstly, we introduce key issues of the experimental environment. Then we compare the memory access workload of stateful applications using different sanitized traffic.

3.3.1 Methodology

3.3.1.1 Traffic Traces

We use real traffic traces as baseline traffic. We take such traces from an OC–3 link (155 Mbps) that connects the Scientific Ring of Catalonia [5] to RedIRIS [68], the Spanish national research network. The Scientific Ring of Catalonia comprises about 40 institutions (research centres and other organizations) connected through roughly 25 access points.

Real Traffic	Sanitized Traffic		
	SN1	SN2	SN3
192.161.35.48	10.0.0.1	10.0.0.1	147.83.53.35
145.32.148.23	10.0.0.2	10.0.1.1	158.109.7.38
192.161.35.48	10.0.0.1	10.0.0.1	147.83.53.35
137.239.42.12	10.0.0.3	10.1.1.1	146.19.5.248
195.232.62.17	10.0.0.4	11.1.1.1	65.167.55.24
131.25.105.36	10.0.0.5	11.1.1.2	211.243.8.86

Table 3.1: Example of resulting sanitized traffic

We develop three techniques to sanitize network traffic that are representative of sanitized traffic traces publicly available. The sanitization only modifies source/destination IP addresses of the packets. Table 3.1 shows an example of resulting sanitized traffic. The top left column provides real IP addresses from the baseline traffic. The remaining columns present the proposed sanitizing methodologies. SN1 uses a counter that increments the last octet of the IP address every time a new address is found. SN2 is similar than SN1, but the incremented octet is selected by round robin. SN3 uses random addresses for every unique real IP address. Most of the publicly available traffic traces are sanitized by the SN1 approach.

Table 3.2 presents the network sites used to compare sanitized traffic between a variety of network bandwidths. The traces are available in the National Laboratory for Applied Network Research (NLANR) [59]. Selected traffic traces show about 25% on average of network utilization and comprise anonimized packets by using methodologies different to the above mentioned.

Label	Site	Link	Theoretical Bandwidth (Mbps)
ANL	Argonne National Laboratory	OC-3	155
MRA	Merit Abilene	OC-12	620
IPLS	Abilene IPLS router instrumentation	OC-48	2480

Table 3.2: Traffic Traces

3.3.1.2 Workload

Most of the benchmark suites for network processing presents layer 2–3 applications: NetBench [54], Commbench [100], NP Forum [60], and NpBench [44]. Only one benchmark initially included in the NetBench suite provides stateful DPI, called Snort [81].

Snort is a Network Intrusion Detection System (NIDS) for real–time traffic monitoring, packet logging, and detecting attacks on a system [10]. The packet processing works as follows: decoding, where the application decodes the packet header and keeps (if necessary) the packet payload; preprocessing, where some ”smart” preprocessors deal with the packet to detect attack attempts; rule–matching, where multiple patterns are checked against packet header and/or content. The stateful processing is found in several preprocessing engines (*e.g.* keeping track the state of connection reliable flows). Our configuration enables stateful preprocessors and the default rule set of 3200 rules.

3.3.1.3 Metrics

We instrument the binary code using the ATOM tool [84]. We use checkpoints to identify start and end of packet processing. Instrumented code generates the memory access footprint as well as the number of memory accesses per packet processing.

Snort only shows processing variations due to IP address distribution in the size of memory access footprint, rather than cache miss rate. The reason behind this is that Snort uses just a few small data structures based on IP address, while there are many other linked lists and tree–based structures based on multiple packet header fields. Thus, cache performance is very similar with sanitized IP addresses, but the memory access workload can significantly differ. Especially when data structures need maintenance (*e.g.* autobalancing of nodes) based on the distribution of flow states (i.e. using the 5–tuple of TCP/IP packets, source/destination IP address, port, and protocol). For this reason, it is

more likely to find variations by comparing footprint size.

Equation 3.2 provides the relative error of the number of data memory accesses between sanitized packets, Mem_{Sanit}, and the packets of the original trace, Mem_{Orig}. In addition, Equation 3.3 calculates the relative error for complete flows. We focus this study on TCP traffic because it represents over 90% of Internet traffic and Snort stateful processing is focused on TCP connections.

$$Packet\ Relative\ Error = \frac{\left|Mem_{Sanit} - Mem_{Orig}\right|}{Mem_{Orig}} \cdot 100\% \tag{3.2}$$

$$Flow\ Relative\ Error = Avg.\left(\frac{\left|Mem_{Sanit_{(i)}} - Mem_{Orig_{(i)}}\right|}{Mem_{Orig_{(i)}}}\right) \cdot 100\% \tag{3.3}$$

We take metrics after a warm up period of 10K packet processing. We collect statistics of 50K packets and 5K complete flows. Larger traffic traces show no significant variations.

3.3.2 Evaluation

Snort uses SplayTree data structures[79] as well as other hash tables. A SplayTree is a self adjusting form of a binary search tree. The tree is autobalanced every time it is accessed. That is, the last accessed node is the new root. The tree keeps the most recently accessed nodes in the upper levels. Flow state nodes are sorted using the 5–tuple flow identification (*i.e.* source/destination IP and port addresses and protocol). Node rotation leads to different distributions depending on the sorting values. When memory has to be released due to the lack of free memory, the first victim nodes are selected from the lower levels, since they are not recently accessed.

Figure 3.3 depicts an example of the memory access workload of a given flow using real packets from the baseline trace (real traffic) and SN1 sanitized packets. The X axis shows the n–th packet of the flow, while the Y axis denotes the number of memory accesses. We can observe that most of the packets present negligible difference between traces. As data structures are optimized, the cost of accessing to a given node is similar for both traces. However, there are few isolated important peaks, due to self–maintenance tasks of the tree. The number of packets that show large memory access

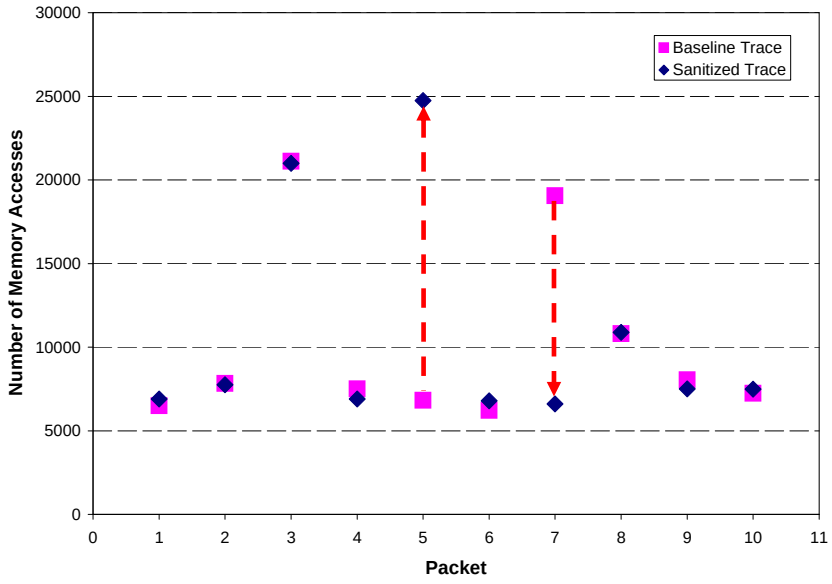
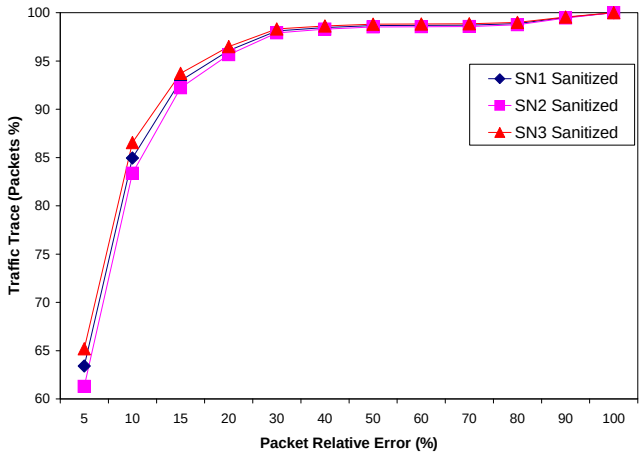


Figure 3.3: Workload of a given TCP flow

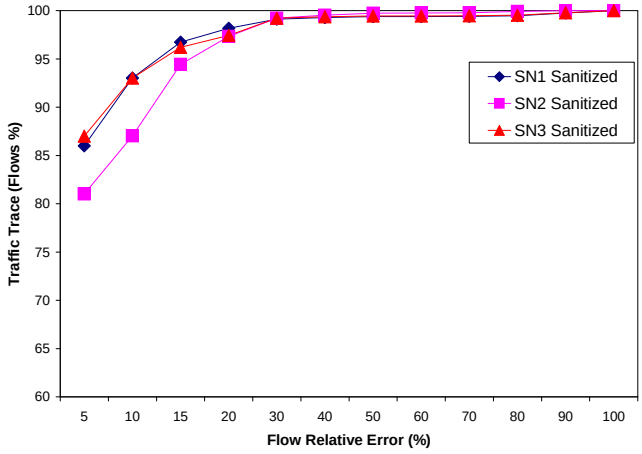
workloads is similar, although they are triggered by different n–th packets. In our example, the baseline trace shows large workloads in the 3rd, and 5th packets, while the sanitized trace shows large workloads in the 3rd and 7th packet.

Figure 3.4(a) and Figure 3.4(b) present the packet and flow relative error of memory access workload, respectively, between the baseline trace and sanitized traces. The X axis indicates the relative error according to the Equations 3.2 and 3.3 for packet error and flow error, respectively. The Y axis denotes the accumulated percentage of the traffic trace.

We can observe in Figure 3.4(a) that the three sanitized traces show similar packet relative error. Using SN3 trace (triangle line) shows about 65% of traffic with less than 5% of packet relative error. SN1 (diamond line) and SN2 (square line) traces presents slightly lower percentage of traffic with less than 5% of packet relative error. In addition, over 95% of the traffic shows less than 20% of packet relative error. That is, most of the packets shows similar number of memory accesses regardless the IP address. In contrast, less than 4% of the packets present significant relative error, due to maintenance tasks executed on different packets. Nevertheless, traces present similar number of workload peaks (see Figure 3.3).



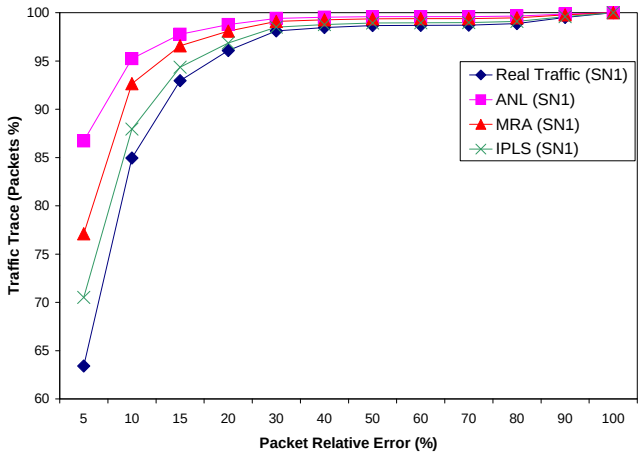
(a) Packet relative error



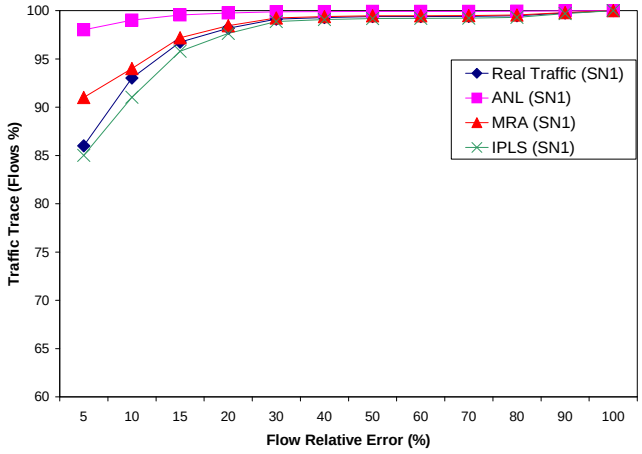
(b) Flow relative error

Figure 3.4: Real versus sanitized traffic

In Figure 3.4(b) we can observe that SN1 and SN3 traces present very similar flow relative error. Over 85% of their flows show a relative error lower than 5%. SN2 trace, however, falls 5 percentage points of flows in the lowest relative error of 5%. Slight differences among packets due to data structure distribution are covered by flows. In fact, all sanitized traces show over 97% of the flows with relative errors lower than 20%. In contrast to the packet relative error, most of the eventual shifted workload peaks are gathered to the same flow.



(a) Packet relative error



(b) Flow relative error

Figure 3.5: Sanitized traffic from multiple bandwidth links

Figure 3.5 shows the relative error of multiple bandwidth links (see Table 3.2). We compare SN1 sanitized traces from ANL, MRA, IPLS sites to the original publicly available traces. The baseline traffic is the real traffic trace employed in the previous experiments. Thus, the SN1 sanitized baseline trace is the same trace plotted in Figure 3.4. The baseline trace shows similar bandwidth than the ANL traffic (OC–3).

We can see in Figure 3.5(a) that ANL traffic show more than 20% of packets compared to baseline trace with relative error lower than 5%. In fact, MRA and IPLS present

higher percentage of packets with the lowest relative error than the baseline traffic. The reason behind this is that the original sanitized traces show more similar IP distribution than the real traffic, but the impact on memory access workload per packet is marginal. Overall, the traces present about 95% of packets with relative error lower than 20%, regardless the bandwidth link. Figure 3.5(b) depicts the flow relative error that manifest higher similarity between flows, regardless bandwidth link. In fact, ANL traffic shows about 98% of flows with relative error lower than 5%.

Above results validate that stateful processing show marginal impact on memory access workload due to sanitization traffic. Moreover, we can see there is no relationship between traffic bandwidth and the impact of sanitization.

3.4 Impact of Traffic Aggregation

The goal of this section is to analyze the impact of traffic aggregation on memory hierarchy performance running network applications. To do this, we propose a mechanism to aggregate traffic from a narrow bandwidth link. In this section we also provide a description of the experimental methodology. Finally, we evaluate the cache performance running applications of multiple network layers.

3.4.1 Traffic Aggregation

In order to measure the impact of traffic aggregation, a particular link must provide traffic traces with multiple aggregation rates. However, this environment is not currently available. Thus, we develop a methodology to synthetically generate traffic by aggregating traffic from a given link. We assume linear network workload scalability, although network utilization depends on many factors (*e.g.* time of the day, social events, users connected). Thus, our methodology scales traffic aggregation as follows: traffic keeps similar unique flow rate while lineraly increases the traffic window size (i.e. number of packets between two packets of the same flow) and, therefore the packet inter–arrival distribution (self–similarity and long range dependency are preserved).

Figure 3.6 depicts the proposed traffic aggregation methodology. Firstly, the original set of traces is filtered to normalize sanitized IP addresses and guarantee independency among sanitized traces. The mechanism merges traces according to the local timestamp (in microseconds) of each trace. According to our experiments, there are few collisions

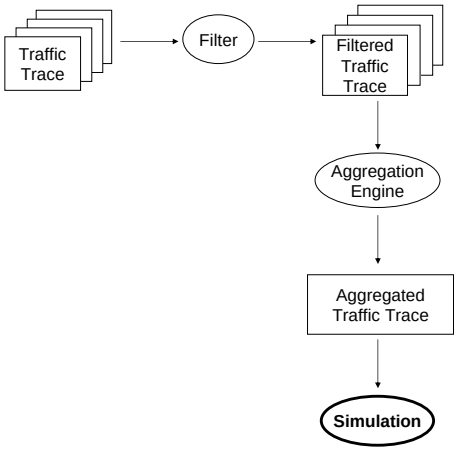


Figure 3.6: Traffic aggregation methodology

due to packets with the same timestamp difference. In these cases an additional mechanism selects the merging order in a round robin fashion. We select the oldest local timestamp and normalize the resulting trace timestamps. Finally, a new sanitized traffic trace presents roughly N times wider network bandwidth than the original link, where N is the number of merged baseline traces.

In Figure 3.7 we can observe an example of aggregated traffic trace that integrates 4 different traces. Each box represents a packet and each pattern means the traffic from a particular traffic trace. The packets are combined according to the normalized timestamp of the trace. Thus, it behaves as a network node with 4 network links.

Our methodology is similar to the proposal of Sagmeister et al. [72] in the sense that we merge a number of traffic traces. However, the approach evenly merges the traces, unlike our mechanism that normalizes the time sequence of the traces. Nevertheless, we don't provide a comparison among both approaches since it is out of the scope of the study of this chapter.

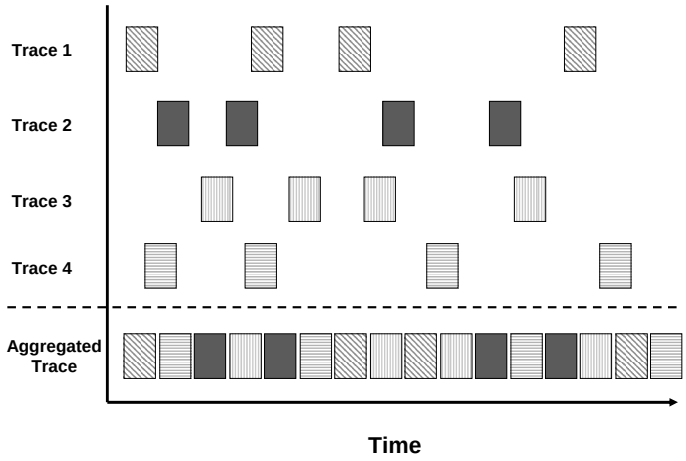


Figure 3.7: Example of aggregated traffic trace

3.4.2 Methodology

3.4.2.1 Traffic Traces

For this study we use publicly available traffic traces. As discussed in Section 3.3.2 stateful processing is not significantly sensitive to sanitization effects [92]. Although in this study we run other types of network applications that can be sensitive to the lost of IP address distribution, there is no other way to do this analysis.

We select a variety of traffic traces from the NLANR site [59] (see Table 3.2) and compare their traffic aggregation in Figure 3.8. The X axis denotes the bandwidth link, such as ANL (OC-3, 155Mbps), MRA (OC-12, 620Mbps), and IPLS (OC-48, 2.4Gbps). The network utilization of these traces is about 25% on average. The left Y axis indicates the average window size (*i.e.* number of packets or unique flows between two packets of the same flow), while the right Y axis denotes the flow unique rate. The gray bars show linear increment of packet window size from one bandwidth link to the other. However, the number of unique flows within a traffic window (the bars with diagonal lines) present non linear increments. As a result, MRA shows over 44% of unique flow rate (the triangle line), unlike the other links that show lower rates. That is, the MRA link presents higher traffic aggregation than the other links, even if it is not the highest

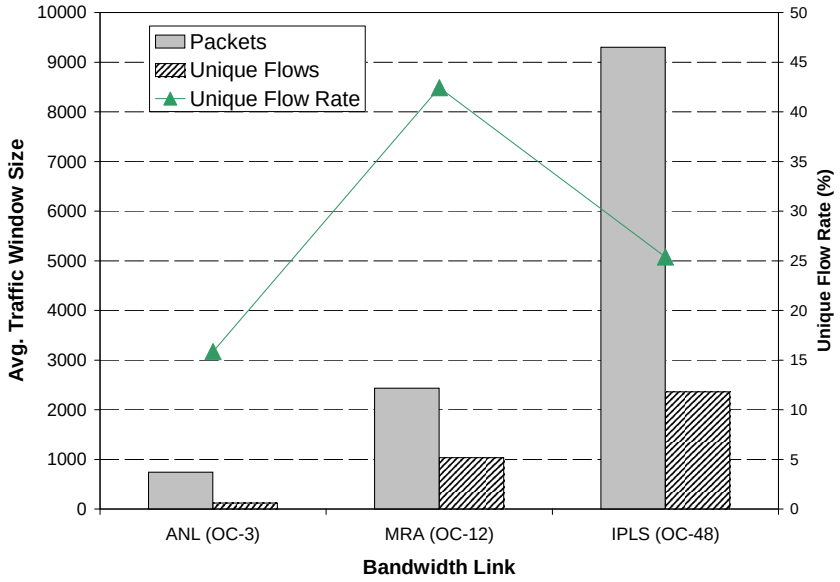


Figure 3.8: Traffic aggregation of several bandwidth links

bandwidth link.

Besides the highest unique flow rate, the MRA link provides the only trace with bi–directional network traffic (*i.e.* packets from source to destination and vice versa). This traffic feature is mandatory for stateful processing. Otherwise flow states can not be correctly updated. Thus, we use MRA as the baseline set of traces to aggregate traffic by using the methodology proposed in Section 3.4.1. Thus, the baseline trace (MRA) presents a real bandwidth of 100 Mbps and we synthetically generate four additional traces of 400 Mbps, 1.6 Gbps, 6.4 Gbps, and 25 Gbps.

3.4.2.2 Benchmarks Selection

We select representative benchmarks from three different network layer processing, as shown in Table 3.3. Layer 2 applications process the packet without external data structures related to network characteristics. For example, CRC only needs the IP packet header. Layer 3 workloads need external data to process the packet. For instance, IP forwarding searches the next hop in the IP forwarding table. Finally, Layer 4 (stateful) applications need information at the transport layer. Stateful applications keep track of

Category	App.	Bench. Suite
Layer 2	AES	NpBench
	MD5	NpBench
Layer 3	NAT	NetBench
	Route	NetBench
Layer 4+ (Stateful)	Snort	NetBench

Table 3.3: Selected Benchmarks

the state of previous packet processing; for example, TCP termination requires to keep the state of the flows, unlike the other applications.

We select two applications that show similar cache performance than the average of every workload category[54, 100, 44]. Regarding Layer 4 (stateful) workloads, there is only one benchmark, Snort[10], which is configured to enable stateful packet processing.

3.4.2.3 Measurement

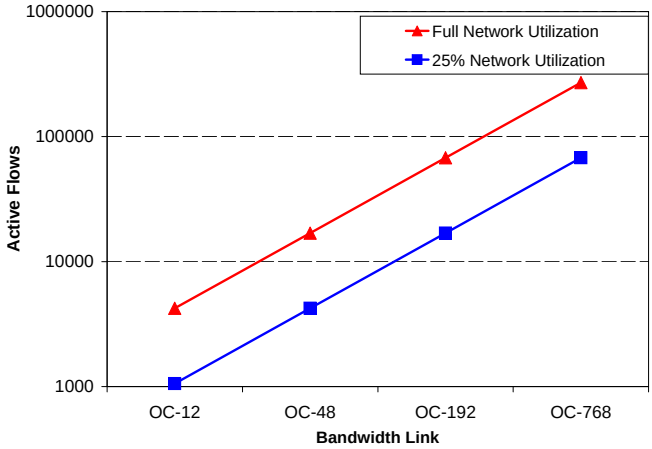
As in Section 3.3.1.3, we instrument the binary code of applications using the ATOM tool [84]. Applications are compiled with -O3 optimization. We use checkpoints to identify start and end of packet processing. The instrumented code generates data memory access traces that are provided to a cache simulator.

We run every benchmark the same number of packets for each traffic trace. We warmed up the environment by processing 10K packets. The memory access traces are generated by processing 50K packets per application. All experiments show stable results with these processing periods.

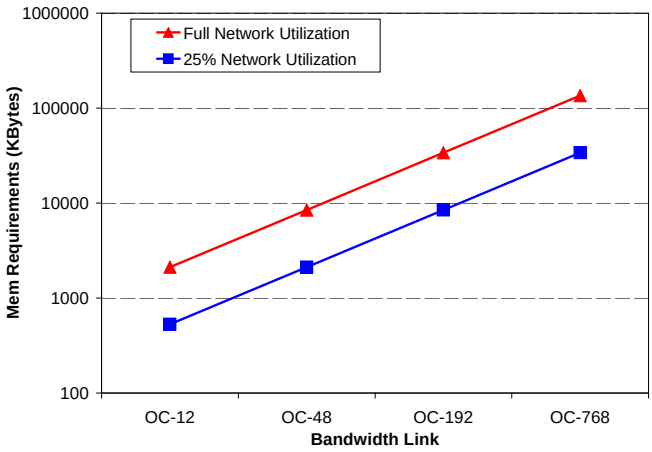
3.4.3 Evaluation

Figure 3.9 depicts a theoretical case of memory requirements for several bandwidth links. The left graph shows the number of active flows within the average traffic window size (Y axis). This study uses metrics of baseline trace (MRA, OC–12) discussed in Figure 3.8. We remark that baseline traffic shows a network utilization of about 25%. We assume linear increment of traffic aggregation for higher bandwidth links as well as for 100% of network utilization (the triangle line). The right graph presents memory

requirements in Bytes to keep stateful data for each active flow. Although Snort can demand larger stateful data structures, we assume a Snort configuration that demands 500 Bytes to point out the memory requirements of stateful applications even presenting lightweight statefulness.



(a) Traffic Aggregation



(b) Memory Requirements

Figure 3.9: Example of theoretical memory requirements

We can observe that a theoretical OC–192 link showing 25% of network utilization (i.e. nearly 2.4 Gbps of traffic out of 10Gbps of bandwidth link) demands about 10MBs

to keep active states for roughly 20K active flows. The data working set can be exponentially higher depending on the state size handled by the application. Current Internet evolution leads to increase traffic aggregation in the network. It is very likely the near future stateful DPI shows larger statefulness. For these reasons, upcoming network links that provide complex stateful DPI will likely demand hundreds or thousands of MBs for sustaining active stateful data structures.

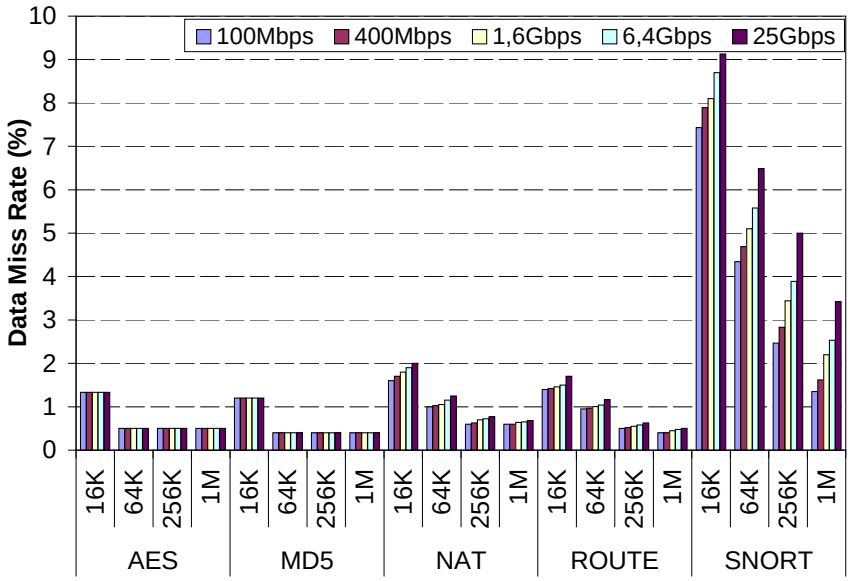


Figure 3.10: Traffic aggregation impact on L1 Data Cache

In Figure 3.10 we can observe the miss rate according to L1 data cache sizes ranging from 16KBs to 1MB running multiple network applications (X axis). The cache configurations assume 4–way set associative with 32Bytes line size. The Y axis indicates the miss rate. We can identify three different behavior that are related to the network layer processing. Firstly, layer 2 applications (AES, MD5) are not sensitive to variations on traffic aggregation, even using reduced cache size. The workload is computational intensive on the packet itself, but there is no need to deal with any external data structure. For this reason, data cache miss rate is negligible. Layer 3 applications (NAT, ROUTER) show a marginal impact. The packet processing deals with external data structures associated to some information of the packet header, such as destination IP address. When the traffic aggregation increases, temporal locality can be reduced depending on the size of external data structures (*e.g.* forwarding table) and the distribution of data searched

(*e.g.* IP address). Larger cache sizes reduces cache miss rate and smooths temporal locality impact.

Finally, SNORT shows considerably higher miss rates and significant impact due to traffic aggregation. The cache miss rate of 64KBs L1 cache size increases from 4,2% to 6,5% as we move from 100Mbps to 25 Gbps. Although larger cache size reduce cache miss rate, similar increments are shown regardless the cache size. The reason behind this is that higher bandwidth traffic shows lower flow state temporal locality on stateful data structures. Thus, it is very likely that memory accesses to a given flow state miss on L1 cache. Under high traffic aggregation, flow state misses can not be overcomed with a larger L1 cache. The impact on L1 data cache depends on both the percentage of flow state memory accesses and the remaining variables handled during packet processing. Snort presents significant impact, even with lightweight stateful processing. For this reason, we suggest that upcoming stateful applications will present a higher impact on L1 data cache performance.

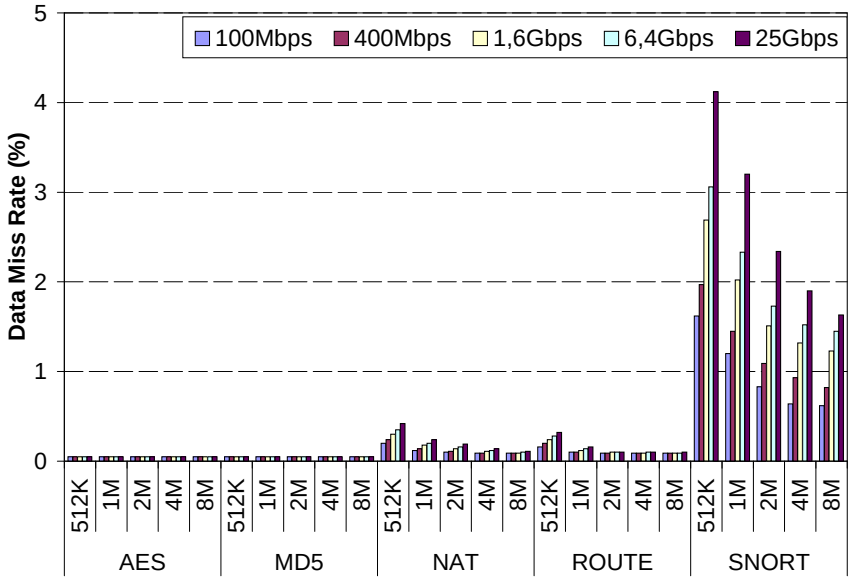


Figure 3.11: Traffic aggregation impact on L2 Data Cache

Figure 3.11 shows L2 miss rate comparing the number of data L2 misses to the amount of memory accesses. We use a L1 cache of 256KB in order to measure traffic aggregation impact without trashing L2 contents due to reduced L1 cache size. We can

observe again three different behaviours among benchmarks. Layer 2 applications show nearly zero L2 cache miss rate, since the data working set perfectly fits on L1 cache, even with reduced size. Layer 3 applications present similar miss rates, but we can observe slight variations with L2 cache of 512KB. The reason behind this is that only reduced locality of data will lead to show L2 cache misses (*e.g.* huge IP forwarding table under high traffic aggregation). In contrast, on systems with large L1 cache, SNORT only accesses to L2 for stateful data. Comparing miss rates of L2 and L1 cache (see 256KB configuration in Figure 3.10) validates that most of the misses in L2 are related to flow state accesses. Due to capacity constraints, L2 cache is likely to be unable to keep all active stateful data structures. The impact of traffic aggregation on L2 cache is roughly 2.5x higher miss rate comparing traffic of 25Gbps to 100Mbps.

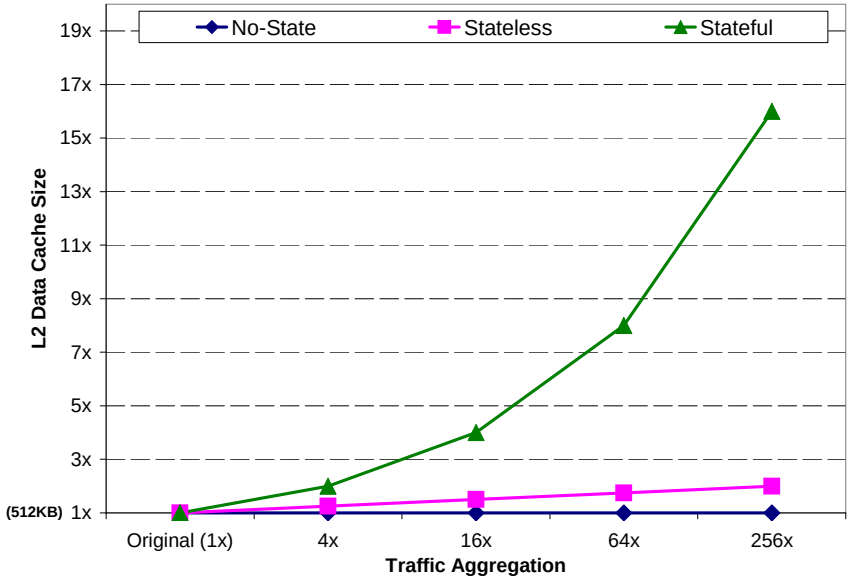


Figure 3.12: L2 Data Cache Requirements

L2 cache size depends on many factors as we have discussed before, such as traffic aggregation, flow state size, percentage and behavior of flow state accesses during packet processing. Figure 3.12 depicts the trend of L2 cache size required to sustain active data structures as the traffic aggregation linearly increases. We average the demands of each application category as described in Section 3.4.2.2. The X axis shows traffic aggregation scales, assuming that Original(1x) is relative to any particular level (*e.g.* 100 Mbps). The effects are the same for 100 Mbps or any other bandwidth link. The

Y axis shows the increment of 512KB L2 cache size (1X). We can observe that Layer 2 applications show the same demand of L2 cache size regardless the traffic aggregation. Layer 3 applications, such as IP forwarding, generally present smoothed linear growth. In contrast, stateful applications show more than linear size increment. The slope can show steep rise running more complex stateful applications.

3.5 Related Work

Research in network processing is very sensitive to network traffic properties. Representative traffic traces are mandatory to present fair studies. Unfortunately, most of the characterization studies of NP benchmark suites use sanitized network traffic traces, such as CommBench [100], NetBench [54], and NpBench [44]. Although detailed workload characterization of layer 2 and layer 3 applications are provided, authors do not analyze any impact of network traffic on the results. Specially consequences due to the use of sanitized traffic.

A large literature on analysis and proposals of sanitizing methods can be found in the state-of-the-art. Kohler et al. [39] demonstrate the loss of Internet IP address distribution through sanitization procedure. TCPdpriv, developed by Greg Minshall [55] and further modified by K. Cho [15], can be viewed as a table based approach. It anonymizes tcpdump traces by stripping packet contents and rewriting packet header fields. However, it may produce inconsistent prefix–preserving anonymization, i.e., same original prefix may be mapped into different anonymized prefixes when independently used on multiple traces.

Crypto-pan [102], a cryptographic algorithm, solves this problem and provides a consistent prefix–preserving scheme. The scheme can maintain a consistent anonymization mapping across multiple anonymizers using a shared cryptographic key. But presents some limits at anonymizing in wire speed. A new approach to transform and anonymize packet traces is proposed by Pang et. al. [62]. It is a tool for packet trace anonymization and general purpose transformation. The tool offers a great degree of freedom and convenience for trace transformation by providing a high-level programming environment in which transformation scripts operate on application-level data elements. Unlike previous packet trace anonymization efforts, packet payload contents are included in the result. The authors verify the correctness of the traces under attack scenarios for pattern matching purposes, but not for stateful purposes. We can see that most of the studies highlight the impact of sanitized network traffic on layer 2 and layer

3 applications due to the variations in the IP address distribution. But there is a lack of knowledge on effects on stateful applications.

Studies have presented analytic models to characterize network traffic. Leland et al. [46] demonstrate the presence of long–range dependence and self similarity in Ethernet LAN and WAN traffic. Other studies evaluate the impact of traffic aggregation on performance, but only from the network point of view. Instead we analyze performance impact from the network processing point of view. Moreover, there is no analysis of network processing applications about the impact of traffic aggregation on memory performance, especially running stateful applications.

3.6 Chapter Summary

In this chapter we have analyzed the impact of two network traffic features on the performance of stateful applications. We have focused our study on IP address sanitization and traffic aggregation. The conclusions of this chapter are applied in the experimental environment throughout this thesis and future work.

We have discussed that stateful applications show very few data structures based on IP addresses, but main data structures are accessed by using hash functions or crossing data tree based structures. Due to this, we have compared the memory access workload per packet between real traffic versus several sanitized versions of the same traffic trace. Experiments from a variety of bandwidth links have been presented. The results show that stateful processing is not sensitive to IP address sanitization. The main reason behind this is that main data structures are based on several packet header values (*e.g.* 5-tuple for TCP and UDP packets). Thus, performance is sensitive to temporal distribution of several header values that are actually preserved by sanitized traffic (*e.g.* port addresses and protocol).

We have proposed a methodology to aggregate traffic in order to generate high network bandwidth traffic traces from lower bandwidth links. As traffic aggregation depends on many network factors (*e.g.* network utilization, behavior of users and applications) it is hard to validate a given scalability trend. For this reason, we have assumed that traffic characteristics linearly scale with bandwidth link. This mechanism overcomes one of the main problems on network processing research. Otherwise, only few publicly available low bandwidth traffic traces show the required features (*e.g.* bidirectional network traffic) to be used as representative data input for stateful processing.

Finally, we have studied the impact of traffic aggregation on data cache performance comparing different types of network applications. Layer 2 applications tend to deal with self–contained packet data as well as external data structures that are not sensitive to network features. Layer 3 applications use to manage data structures related to network features (*e.g.* IP address distribution). Therefore they can show lower cache performance using very large data structures, although it depends on the network property locality. Nevertheless, the impact is generally marginal for applications like IP forwarding. Unlike stateful applications that shows significant impact even running lightweigth stateful processing, since they need to keep states of active network connections. We have shown that the more traffic aggregation the more memory requirements and the lower temporal locality of flow states. Overall, the impact of traffic aggregation depends on the application itself and on the data structures distribution.

We have not include network traffic characterization, since it is out of the scope of the thesis. However, Section 7.2 proposes some future work on network traffic modeling and correlation between network characteristics and stateful workload.

Chapter 4

Characterization of Networking Applications

The trend of network processing is to increase the intelligence of the routers. This means that there is an increment in the workload generated per packet and emerging types of applications, such as stateful network services. In addition, as Internet traffic continues to vigorously grow, traffic features, such as network bandwidth and traffic aggregation, increase even more the stress on traffic processing, overloading the capacities of the systems.

The overall goal of this chapter is to characterize workloads of network applications. The results of the studies will be the basis for proposals to improve packet processing. First, we classify the workloads according to the data management process. We analyze the workload characterization of a representative set of applications of each category. The analysis emphasizes the study of data cache behavior. We also characterize other issues, such as branch prediction, instruction mix, and Instruction Level Parallelism.

We deeply study the impact of stateful applications on architectural bottlenecks along the life of a given network connection. The memory impact is related to the statefulness of the application itself. Moreover, depending on the target of the application, the memory bottleneck may be concentrated within a set of packets or distributed along the TCP connection lifetime.

The results show an important memory bottleneck that involves new challenges to

overcome, especially for oncoming parallel architectures.

4.1 Chapter Roadmap

Section 4.2 presents the proposed classification for network applications according to the data management and processing. Section 4.3 presents the evaluation methodology employed in the experiments of this chapter.

The analysis is performed in two steps. First, we characterize every workload category in Section 4.4 showing important differences among them. Then, we deeply study the characterization of different stateful workloads throughout the life of TCP connections in Section 4.5.

We conclude discussing related work and summarizing the chapter in Section 4.6 and 4.7, respectively.

4.2 Workload Classification

In general, network processing is classified as control plane and data plane [44]. The former is focused on the management of the communications (*e.g.* quality of service) and processing of the packets is not ”on the fly”. On the contrary, the latter is focused on the network processing itself (*e.g.* packet forwarding) which is critical to the traffic flow rate.

Control and data plane classifies packet processing, but network applications show different workloads according to the data management. Figure 4.1 depicts our classification proposal to categorize the following workloads:

- *self–contained*: programs that only need the packet itself to perform the packet processing. There is no need to search additional data related to the packet or connection. For example, CRC only needs the IP packet header.
- *stateless*: applications that generate no record of previous packet processing. Unlike self–contained category, stateless applications search information to be checked against the packet header and/or payload (*e.g.* IP lookup data structures, pattern–matching). Thus, they can demand large amount of memory for shared

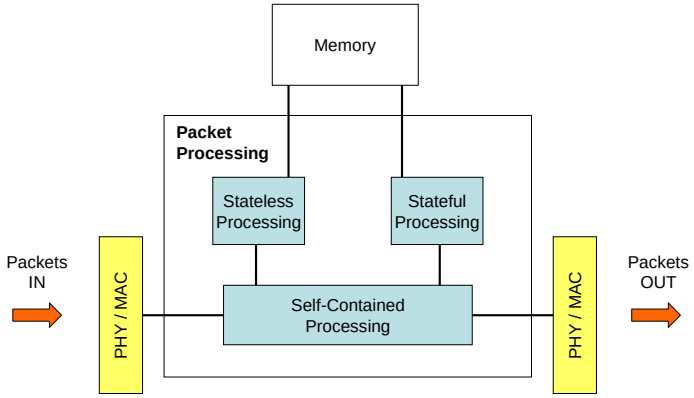


Figure 4.1: Network Processing Workload Categories

data structures. Nevertheless, there are negligible dependencies among packets, since structures are occasionally updated.

- *stateful*: programs that keep track of the state of packet processing [53], usually by setting fields of state related to the flows or connections. For example, TCP termination requires to maintain the state of the TCP flows. The main difference between stateful and stateless programs is that the former may update a variety of fields within the state. Instead stateless applications only require the value and do not update any information.

Most of the low layer applications show self–contained and stateless packet processing, since only basic operations are applied to the packet. In fact, deep packet inspection integrates stateless processing (*e.g.* pattern–matching of payload). Layer4 and upper layer applications can present stateful packet processing to provide additional knowledge about previous packet processing. In general, complex packet processing, such as stateful DPI for security issues, integrates all above workload categories.

4.3 Environment and Methodology

4.3.1 Traffic Traces

In order to perform a fair comparison among applications from different benchmark suites we cannot use the default traffic traces included in the suites, since the traffic is not representative according to the properties discussed in Chapter 3.

We select a public traffic trace from the NLANR site [59] with bidirectional traffic. We apply the traffic aggregation methodology proposed in Section 3.4.1 and generate a traffic trace that simulates a link of nearly 1Gbps with 20K active flows on average. Since the stateful applications deal only with the flow state, the provided aggregation level is enough to represent the low locality of flow states of larger network links.

While the resulting synthetic trace preserves sanitized IP addresses, stateful processing is not sensitive to sanitization [92], as we validated in Section 3.3.2. The selected self–contained applications are not sensitive as well, because they don't need the IP address to perform any search processing. Instead, the stateless applications that we use are affected by sanitized traffic. But, they show no significant variations in the memory performance. Moreover, currently there is no better way to perform this analysis.

4.3.2 Benchmark Selection

We select different sets of benchmarks for the two parts of this Chapter. The first study employs a set of applications of every workload category presented in Section 4.2. We can observe in Table 4.1 the selected benchmarks representative for each workload category. The column "Workload Category" specifies the workload category associated to the benchmark; the column "State Categories" shows the states handled by the application; the last two columns specify the names of the benchmark and benchmark suite. We can observe that there are several state information categories: packet (Pkt), network connection (Flow), global information (Global), and application (App).

The lack of stateful applications within the publicly available benchmark suites involves that there is only a single application that presents stateful processing: Snort 2.4 [10]. We employ different configurations depending on the enabled features. *Snort_SLess* is configured to execute stateless preprocessors. The rest are tuned to use the stateful preprocessors, called Stream4 and Flow Portscan. *Stream4* performs inspection of establishing TCP connections and their maintenance to prevent attacks such as

Snot [82] and Stick [20]. *Flow–portscan* is an engine designed to detect portscans based on flow creation and the goal is to catch one–to–many hosts and one–to–many port scans. In addition, we select *Argus* [6] (*i.e.* network Audit Record Generation and Utilization System) even it is not included in any benchmark suite. This application is a fixed-model Real Time Flow Monitor. That is, it can be used to monitor individual end systems or activity on the entire enterprise network.

Workload Category	State Categories	Benchmark	Bench. Suite
Self–Contained	Pkt	AES	NpBench
	Pkt	MD5	NpBench
Stateless	Pkt & Global	Route	NetBench
	Pkt & Global	Nat	NetBench
	Pkt & Global	Snort_SLess	Snort
Stateful	Pkt & Flow & Global	Stream4	Snort
	Pkt & Flow & Global	Flow–portscan	Snort
	Pkt & Flow & Global	SfPortscan	Snort
	Pkt & Flow & Global	Merged Engines	Snort
	Pkt & Flow & Global	Argus	Argus

Table 4.1: Benchmark Classification

As the second part of this Chapter is completely focused on stateful workload characterization, we include additional stateful configurations of Snort, besides Stream4 and Flow–portscan: *SfPortscan* detects portscans in order to detect the different types of scans Nmap (*i.e.* the most common port scanning tool in use today) can produce; *Merged Engines* is a configuration that enables all above stateful configurations of Snort, that is Stream4, Flow-portscan, and SfPortscan. The combination of engines doesn't necessarily lead to a linear aggregation of the workload. There are engines that can share information (e.g. flow state) and, therefore, preprocessors can collaborate in order to create more robust application.

4.3.3 Evaluation Methodology

We use different tools for taking statistics through instrumentation and for measuring performance through simulation, as shown in Figure 4.2. We instrument the binary code with ATOM [84] and generate statistics for instruction distribution. The applications have been compiled with -O3 optimization.

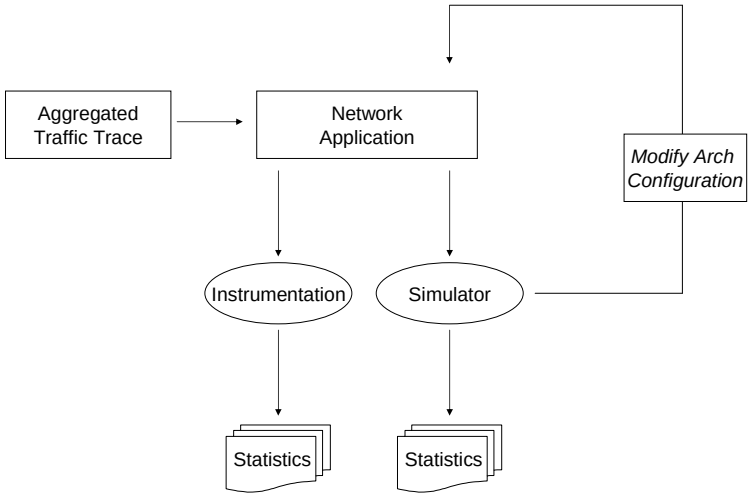


Figure 4.2: Evaluation Methodology

We use a modified version of the SMTSim simulator [89] to simulate a single threaded out–of–order architecture with the baseline configuration shown in Table 4.2. The baseline features a high–performance configuration in order to measure the actual application behavior and to avoid any architectural limitation. Some experiments modify the configuration to study the impact of different architectural bottlenecks, such as data cache hierarchy and branch prediction.

We run every benchmark using the selected traffic traces and processing the same number of packets. Before starting to take statistics, we run the applications until the initial stage is finished, such as the creation of IP lookup table. The applications are warmed by running enough packets in order to reach the stable behaviour of the program. Our studies indicate that 10K packets are enough for the warming stage. Statistics are collected by processing 25K packets to obtain representative results. Longer simulations show negligible variations.

Processor Configuration	
Fetch Width	4
Queues Entries	64 int, 64 fp, 64 ld/st
Execution Units	6 int, 3 fp, 4 ld/st
Physical Registers	384 int, 384 fp
ROB Size	256 entries
Branch Predictor Configuration	
Branch Predictor Perceptron	256 perceptrons, 4096 x 14 bit local 40 bit global history 6 cycles miss penalty
Branch Target Buffer	256, 4-way
RAS	256 entries
Memory Configuration	
ICache DCache	64KB, 4-way, 8 banks, 32B lines, 1 cycle access LRU replacement policy
L2 Cache	2MB, 8-way, 16 banks, 32B lines, 20 cycles access LRU replacement policy
Main Memory	500 cycles access

Table 4.2: Baseline Configuration

4.4 Characterization of Network Workloads

4.4.1 Stateful Data Requirements

Network applications process packets handling different types of information. The basic data structures needed is the packet related data structures. Every packet manages its own data structure that is completely independent to the other packets. Those include most of the data structures employed by the self-contained applications, as opposed to stateless and stateful applications, which need to access other type of information. The data only covers the current packet processing, such as packet content. The lifetime of these data structures is very short, since they are contained within each packet processing. The inter–packet temporal locality of this data set is independent of any network

traffic parameter.

Networking applications usually handle a global state. Actually, the data can be classified in two subsets: global data structures that are independent of any network traffic parameter, such as total number of processed packets, and global network data structures, such as IP lookup table, related to a particular network traffic property, such as IP address distribution. Thus, the temporal locality of these data structures among packets is determined by the network traffic features. The behavior of those data structures are critical for the performance of stateless packet processing. In addition, the lifetime of global data structures is as long as the lifetime of the application, unlike the packet–processing dependent lifetime in packet data structures.

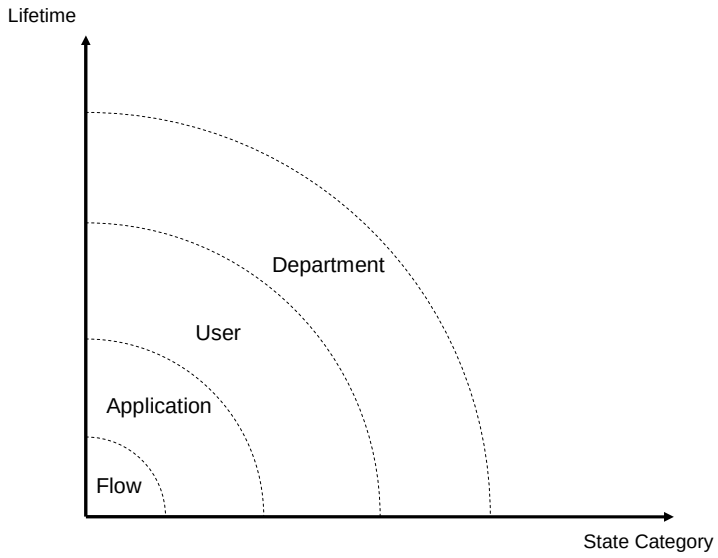


Figure 4.3: Lifetime of State Categories

Nevertheless, network applications with statefulness show additional categories of data according to the granularity of the state managed by the application. Figure 4.3 depicts several state categories (X–axis) and their lifetime (Y–axis):

- **Flow:** The data structures are related to the network connection, such as flow state. The temporal locality is determined by the traffic aggregation level of the network link and the burstiness of the traffic. Although the lifetime is longer than packet state data structures, it is delimited by the lifetime of the flow. For example, most

of the web sites connections exchange from 10 to 20 packets per object according to the HTTP protocol.

- **Application:** The data structures are associated to the application layer, such as Real Time Protocol (*i.e.* a transport protocol designed to provide end-to-end delivery services for data with real-time characteristics). A similar granularity state level could be defined as macro–flow, such as counters of a set of flows, which is used for traffic monitoring/management. The temporal locality is determined by the traffic aggregation level. However, the lifetime of the data is determined by the group of flows.

- **User:** It includes all information regarding a given user. This state level is very common on network monitoring applications, although it is employed in other areas such as VoIP and policy management. A given user executes a number of applications sending/receiving packets from many flows. The temporal locality is determined by the traffic aggregation level, although it is more likely to increase when the state belongs to upper levels. The lifetime of the data is determined by the user and application requirements.

State categories that present longer lifetime also show higher probability that two packets related to the same state are close together. In fact, it is very likely than future applications present other states (e.g. office, department, company) as they provide services related to those states. The behavior and locality properties depend on the traffic aggregation level and the state itself.

Figure 4.4 depicts the flow state kept by the studied applications. We can observe the difference between any stateless program and the rest of stateful application. The Merged Engines configuration shows the largest flow state with 1.5 KByte. SfPortscan shows approximately 1 KByte of state, although some flows only require roughly 600 Bytes. Argus present a flow state base of over 500 Bytes. However, depending on the configuration the state can be quite more than 1KByte. We use a configuration that activates, at the most, 1KByte of state.

4.4.2 Instruction Mix

The applications show nearly 50% of instructions are arithmetic, shift and logic operations, regardless the workload category. Most of the applications presents similar branch operations ranged from 10% to 20%. Only *AES* shows a lower percentage. Finally, an

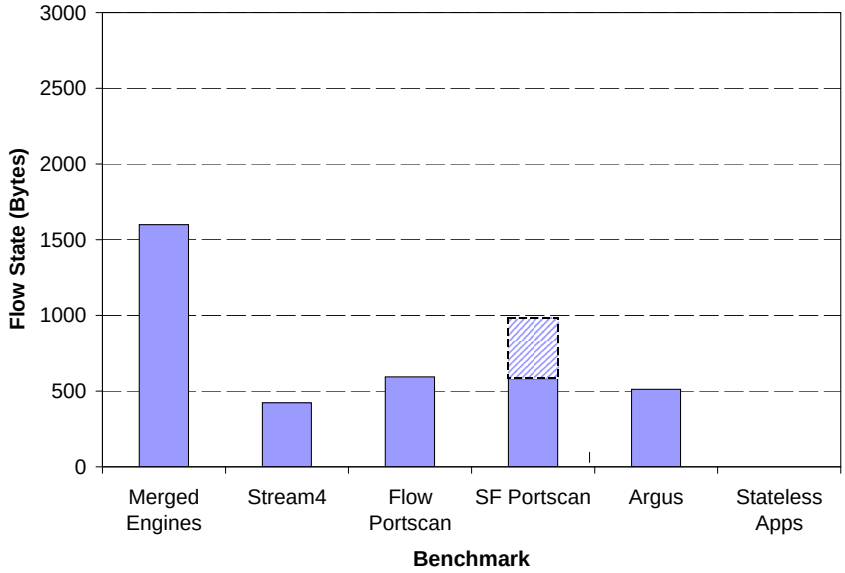


Figure 4.4: Statefulness of Benchmarks

average of 40% of instructions are memory accesses, which more than 60% are loads. The instruction distribution shows minor variations according to the application categories. Nevertheless, both stateless and stateful applications are slightly more memory stressed, due to the search of additional data, such as IP lookup or flow state.

The amount of instructions per packet depends on the application itself rather than the workload category. For example, AES, MD5, NAT, and Route benchmarks present from about 100 to nearly 500 instructions per packet. However, Snort_SLess (*i.e.* intrusion detection with only rule–matching engine) show from hundreds to thousands of instructions depending on the size of rule set.

We can observe in Figure 4.5 that stateful processing presents large workloads that ranges from about 3K to nearly 7K instructions per packet. The Y–axis denotes the number of instructions per packet, while the X–axis indicates the stateful benchmark. The greatest part of the processing is covered by nearly 45% of integer computation and 35% of memory accesses. The rest of the workload consists of about 12% and 8% of conditional and unconditional branches, respectively. Argus is more memory intensive showing approximately 45% of memory accesses and less conditional and unconditional branches with about 7% and 3%, respectively, basically due to the monitoring purpose

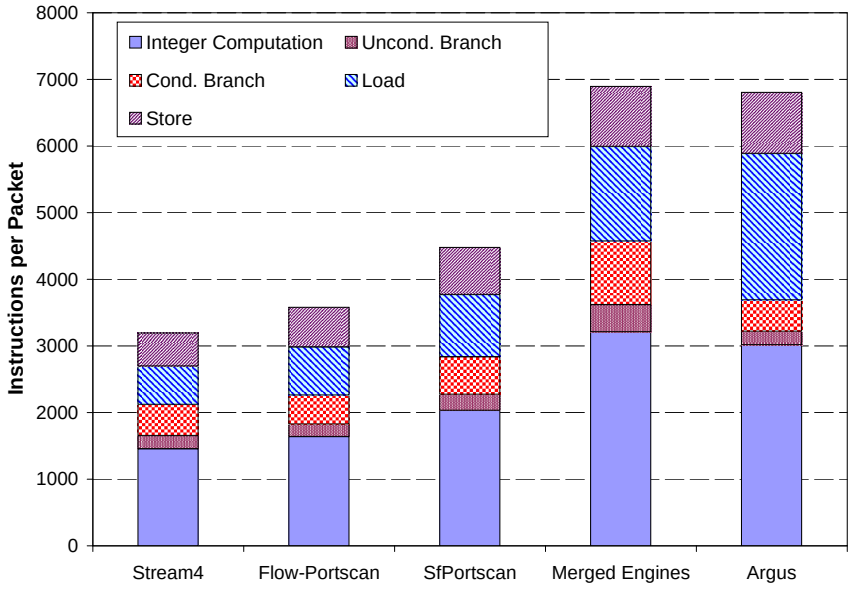


Figure 4.5: Instruction Mix of Stateful Benchmarks

of the application.

The results point out that the computational workload of stateful processing is independent of the packet size, unlike other DPI applications (e.g. pattern–matching of packet payload). In contrast, the selected benchmarks show a relation between the size of the flow state and the computational workload (see Figure 4.4). Nevertheless, further analysis should be done to validate this relationship.

4.4.3 Instruction Level Parallelism

The processor configuration used in this Section presents variations over some of the baseline parameters (see Table 4.2) towards avoiding any additional performance constraint and being able to evaluate the instruction level parallelism (ILP) of the applications. There are no limitations on both fetch bandwidth, instruction cache line size, and functional units. The new configuration also presents an oracle branch predictor and a perfect memory system, where all predictions are hit and every memory access has one cycle latency, respectively.

Figure 4.6 shows the available ILP of the applications as a function of the inherent data dependencies and data flow constraints. The Y–axis denotes the IPC. The X–axis indicates the benchmarks as well as the workload categories. We can observe that the ILP is independent of the workload category, since it is inherent to the application itself. For example, *MD5* presents an IPC of 3.5 against the 4.7 of *AES*, even though both of them belongs to the same category. In fact, Lee et al. [44] show that Control Plane applications show significantly higher ILP than Data Plane applications, regardless our workload classification. The authors claim that Control Plane applications have more opportunity to exploit ILP than Data Plane processing. The benchmarks used in our studies are Data Plane applications.

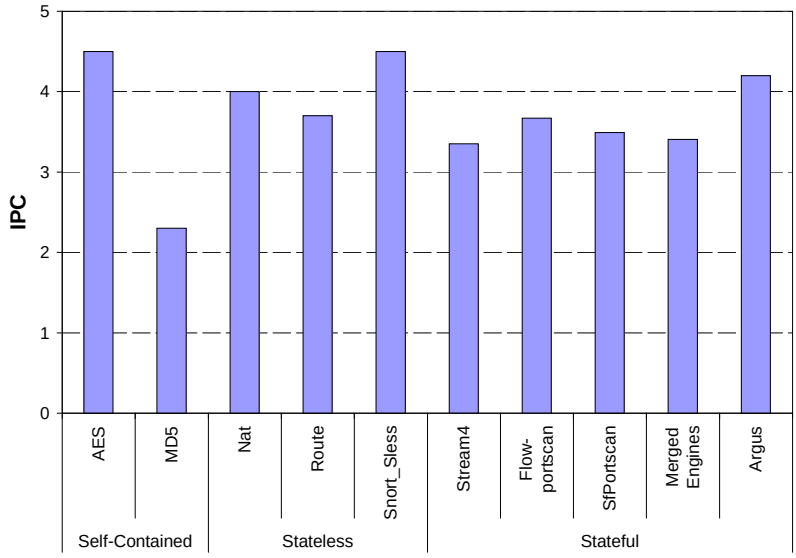


Figure 4.6: Available Instruction Level Parallelism

The stateful applications present an ILP of 3.5 on average, slightly lower than stateless applications that shows about 4 on average. The data flow of stateful packet processing presents high rate of dependencies and therefore the ILP is lower. Most of the packet processing requires to search a state (probably it is not in the cache), check the value, and trigger an action or modify the data structure depending on the value of the state. Thus, the processing workload exposes large percentage of long latency instructions (main memory access due to stateful data) and a significant rate of dependencies among instructions. Therefore, techniques to exploit the ILP are not cost effective for improving the performance of stateful programs.

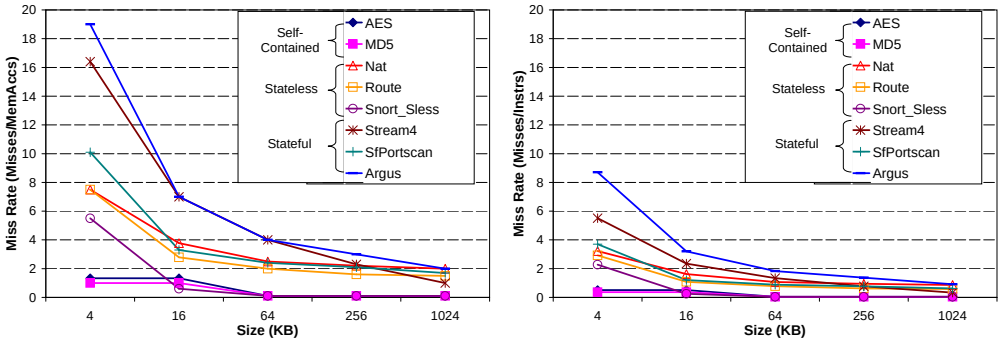
4.4.4 Data Cache Behavior

In this section we discuss the data cache behaviour of the evaluated applications and analyze the impact of data access distribution shown in the above section. We do not include an analysis of instruction cache. As other papers explain [100, 54, 44], network applications present on average near to 100% of instruction cache hit rate, since they are simple applications executed in single threaded. Moreover, our experiments show similar results with selected stateful applications. Other more complex and/or multithreaded network applications can arise important problems in the instruction cache. This issue is studied in Chapter 6.

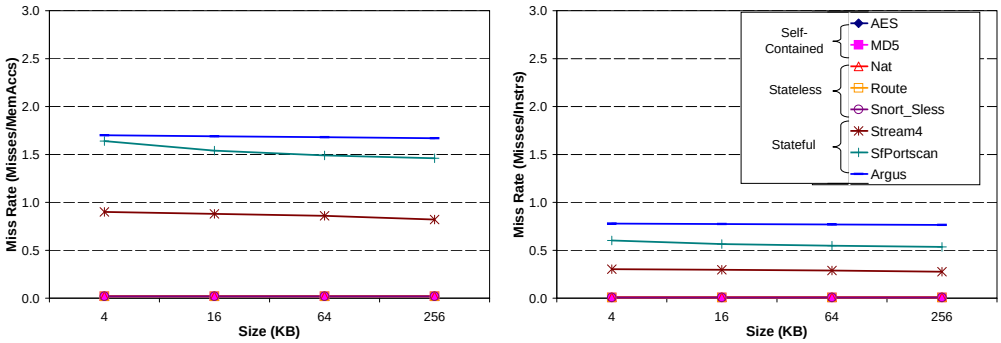
Figure 4.7(a) shows the L1 data cache miss rate using a variety of sizes, with 4–way set associativity and 32B line size. The X–axis shows the sizes of L1 data cache. In the legend of the graph we group the benchmarks according to the application category. We can observe that the self–contained applications present very reduced data cache miss rate, even with reduced size data caches. Most of the working set is related to the packet state. Thus, hit rate is high since a reduced cache is able to succesfully keep this type of structures. The working set of stateless applications is larger due to global data structures, such as IP lookup table. As access rates to these structures are higher, reduced caches present higher miss rates. However, a cache size of 16 KBytes or higher presents a low data cache miss rate of 2% on average. On the other hand, *Snort_SLess* could present higher miss rates if it is a rule-based IDS and there is a large number of rules. Finally, stateful applications present higher miss rates than the rest of applications, due to larger working set and data structures sensitive to network traffic properties, such as distance between two packets of the same flow.

We analyze the effects of increasing the associativity level, but there are no significant benefits. Because the main problem is not cache conflicts, but the cache inability of maintaining the data of the active flows. However, using a larger cache line size we take advantage of spatial locality within the packet processing itself, but it depends on how the application processes the data. For example, if we enlarge from 32B up to 64B cache line size we achieve a 3% reduction on average in the cache miss rate. This fact corresponds to three times the improvement obtained by doubling the cache associativity degree.

The main impact of statefulness resides in the L2 data cache miss rate shown in Figure 4.7(b). While self–contained and stateless applications present near to zero miss rate, stateful applications show a saturated miss rate from 1% up to 1.7%. Even using large L2 data cache the misses due to stateful data structures cannot be reduced. That is,



(a) $DL1 Miss Rate



(b) $L2 Miss Rate

Figure 4.7: Data Cache behavior

almost 100% of DL1 misses also are misses in L2 when we use a very large DL1 cache.

Depending on the size of flow-states and the traffic aggregation level, we can require several MBytes of data structures for flow-states. In fact, the memory performance of stateful applications is very sensitive to the traffic aggregation level [93], since the memory capacity requirements significantly grows and the temporal locality of flow state is reduced. Obviously, with larger flow-states the sensitivity is higher. Stream4 and Argus requires roughly 420 Bytes and 1 KByte, respectively. As we mentioned in Section 4.3.1, we handle roughly 20K flows on average. Thus, the memory requirements of Stream4 and Argus are roughly 8.4 MBytes and 20 MBytes, respectively. Larger network links demand hundreds of MBs (see Figure 3.9). This requirement is higher on applications with more statefulness rates.

4.4.5 Branch Prediction

In Section 4.4.2 we analyze the instruction mix of the selected benchmarks, which present 12% on average of branch operations, which 70% are conditional branches.

In this section we evaluate the behavior of the branches through the study of branch prediction accuracy. We use a perceptron predictor [36, 95]. We also study other branch predictors, such as g-share [51] with different PHT size configurations. The experiments show slightly worse results than the perceptron predictor (less than 3% of difference), with marginal variations as a function of the PHT size.

Figure 4.8 shows the branch prediction hit rate of the perceptron predictor. Overall, hit rates are higher than 96%. In fact, self–contained applications present the highest rates, since the behavior of branches are similar among packets. The results also show that there is no relation between prediction hit rate and flowstate size (see Figure 4.4).

The branches of stateful processing, such as Stream4, are sensitive to network properties (*e.g.* traffic aggregation), unlike the self–contained and stateless applications. The reason behind this is that the action depends on the state related to the packet. Thus, there is a negative aliasing among independent packets. Instead, Flow–portscan and Argus present lower negative aliasing, since the behavior is similar among independent packets, regardless the flow state.

4.4.6 Performance Evaluation

In this section, we use four different configurations: the baseline configuration, an oracle branch predictor, a perfect memory system (*i.e.* every memory access has one cycle latency), and an oracle predictor with a perfect memory system. We remark that the baseline configuration is based on the architecture described in Table 4.2. That is, an out–of–order 4–width superscalar processor, with 64KB of IL1$ and DL1$, and 2MB of unified L2$.

Figure 4.9 presents the IPC (Y–axis) of every benchmark according to the workload category (X–axis). We can observe that self–contained applications present marginal differences on IPC among architectural configurations. In contrast, stateless applications experience a slight improvement moving from about 2.6 IPC (blue bar) up to 3.2 IPC (bar with horizontal green pattern). However, the other stateless applications (Snort_SLess) shows marginal improvement, because it shows slightly lower data cache miss rate as well as lower branch miss prediction. In fact, stateless applications can be sensitive to

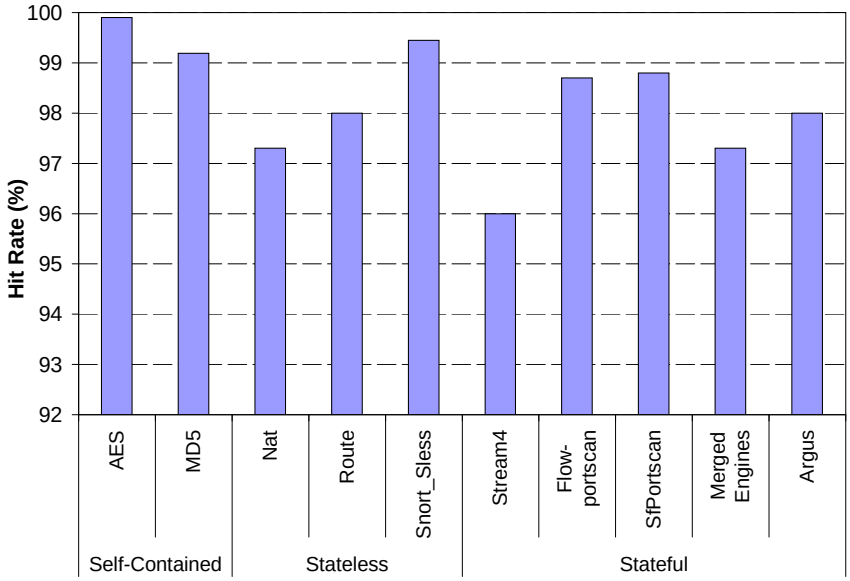


Figure 4.8: Branch Prediction Hit Rate

variations on large data structures (see Section 3.4.3), however it is out of scope of this thesis to study stateless applications.

Stateful applications emphasize the impact on memory performance. The cache performance is stressed since neither DL1 nor L2 cache are able to keep the states of the active flows. We can observe that Merged Engines shows a very reduced IPC of 0.12. The other stateful benchmarks present higher IPCs, but all of them are lower than 0.9. The IPC of the benchmarks should tend to be inversely proportional to the flow state size of the applications, due to the limitations and bottlenecks previously discussed (*e.g.* negative aliasing on data cache and branch prediction, low ILP). However, we can observe that SfPortscan shows better performance than Flow-Portscan, even similar to Stream4. The main reason of these results is the reduced statefulness of the benchmarks, since they reduce the negative impact of statefulness.

Figure 4.10 clarifies the results discussed above by showing the normalized speedup (Y–axis) compared to the baseline configuration. We can see that self–contained applications shows less than 2% of speedup using perfect memory and oracle branch predictor. Stateless applications increase the speedup up to nearly 20%. However, the highest speedups are experienced running stateful applications. While perfect branch predictor

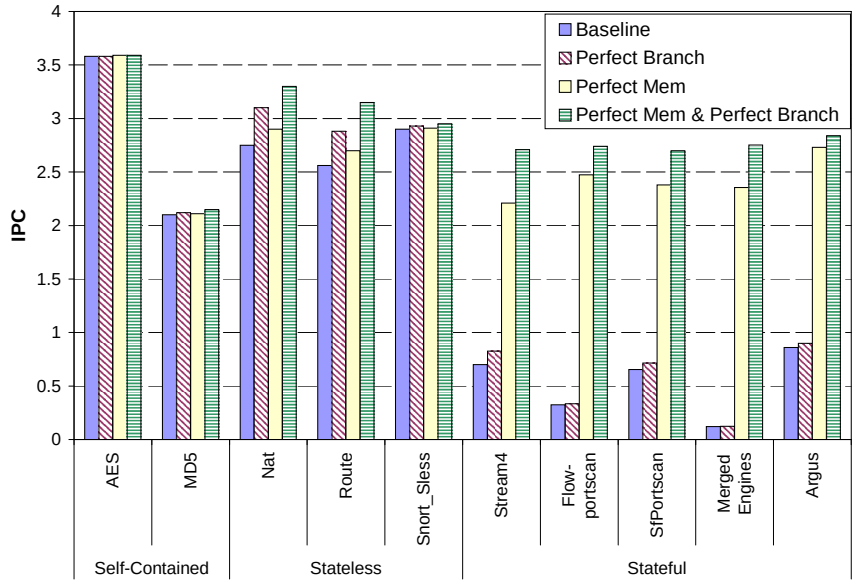


Figure 4.9: Performance Impact of Architectural Bottlenecks

improves the IPC about 10% on average, the perfect memory configuration show larger speedups of 3x. In fact, Merged Engines shows the highest speedup with 22.7x using perfect memory and oracle branch predictor.

4.5 Characterization of Stateful Workloads Throughout a Flow

The previous section presents the differences among network processing workload categories, using averaged measurements among packets. Instead, this section deeply analyzes a variety of stateful workloads throughout the life of TCP connections. That is, a TCP flow with the three–way handshake connection stablishment (*i.e.* SYN, SYN–ACK, ACK packets), data transfer, and the two–way handshake connection termination (*i.e.* FIN, ACK packets). In these experiments, we collect metrics of flows of 10 packets size (*e.g.* a typical HTTP transaction).

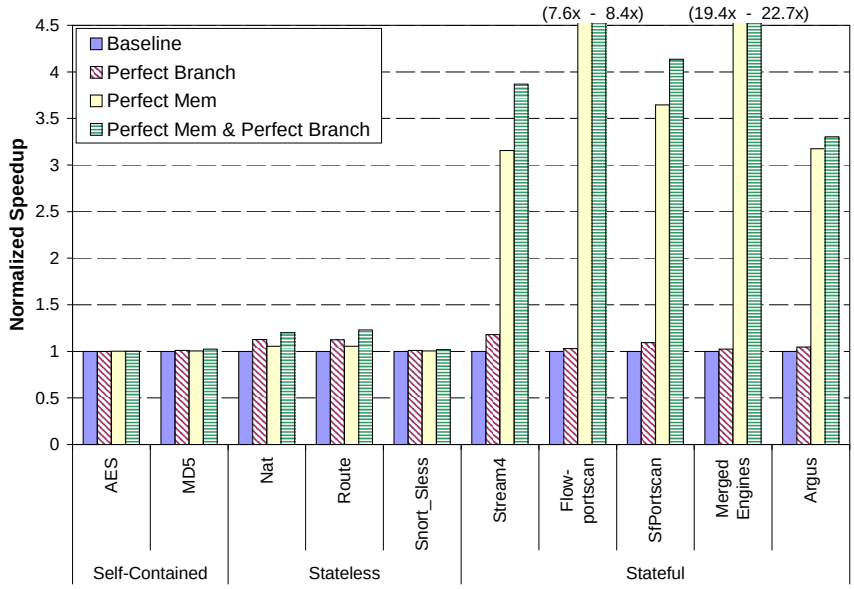


Figure 4.10: Normalized impact on IPC of Architectural Bottlenecks

4.5.1 Computational Workload

Low layer applications, such as IP forwarding, use mainly a set of selected bytes from the packet header to proceed the packet processing. The computational workload is similar across all packets. Other applications that process more information of the packet, such as pattern matching in a DPI application, shows a direct relation between packet payload size and the number of instructions executed per packet. The larger packet size the more comparisons have to be done to find the keyword. This relation may slightly vary depending on the processing algortihms and other features of the application. Finally, stateful applications associate the computational workload to other characteristics, such as state size and the computational requirements of the states. Therefore, stateful applications can show heavyweight processing with short packets and lightweight workloads with large packets.

Figure 4.11 depicts the average of instructions executed by the n-th packet processing through the flow lifetime. The X–axis indicates the n–th packet and the transport protocol stage. As the data transferring stage may be longer, it is represented with a dot line. The measurements are similar among packets during the data transfering. The

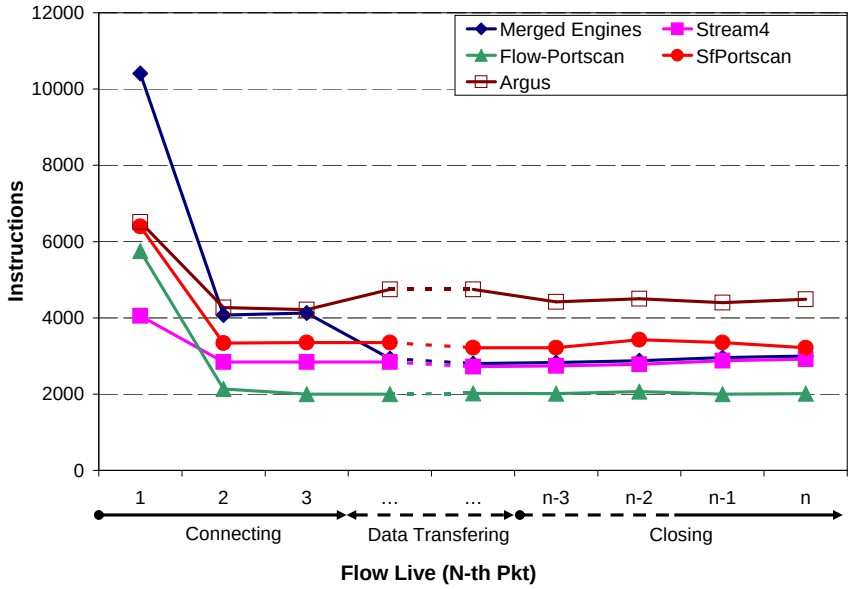


Figure 4.11: Workload per packet during the flow lifetime

results point out that the instruction mix is similar among packets, with the exception of the first packet of a TCP connection, which shows nearly 3% higher memory access rates, due to the initialization of data structures.

We can observe that the Stream4 and Argus show an almost constant behavior in every packet due to the constant task through the flow lifetime. However, the other benchmarks focused on security issues present lightweight packet processing when the application detects that a given packet belongs to a flow previously identified as a safe connection. In fact, we can see that once a connection is stablished Merged Engines generate lower workload than SfPortscan, although the former manages almost double flow state size than the latter. The Merged Engines comprises several preprocessors that can share state leading to a more robust application. On the contrary, SfPortscan requires to execute more instructions, because its engine is less ”intelligent” due to the lack of additional state. Overall, if the application doesn’t have to do regular tasks on every packet processing (*e.g.* monitoring), the hardest workload is concentrated on the first packets of a connection.

4.5.2 Data Cache Behavior

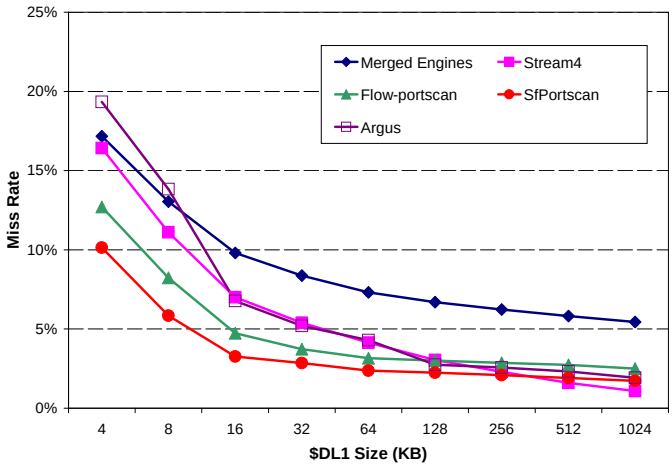
Figure 4.12 shows the data cache behavior using different sizes (X–axis). The results complement the results discussed in Section 4.4.4. We isolate the stateful applications to understand better the variations among different stateful processing. Figure 4.12(a) shows the DL1 miss rate. Increasing the cache size significantly reduces the miss rate with reduced caches. Caches larger than 64 KBytes do not present important miss rate reductions. Merged Engines shows more than 5% of cache misses even using 1MByte.

We can conclude that a DL1 cache of roughly 64 KBytes is able to maintain the variables that present temporal locality between packets. These data are not sensitive to the variations of network traffic characteristics discussed on Section 3.2: structures to temporally hold contents of the packet itself (*e.g.* packet header and payload); stateless data shared among packets (*e.g.* global packet processing options); and coarse grain stateful data (*e.g.* global counters).

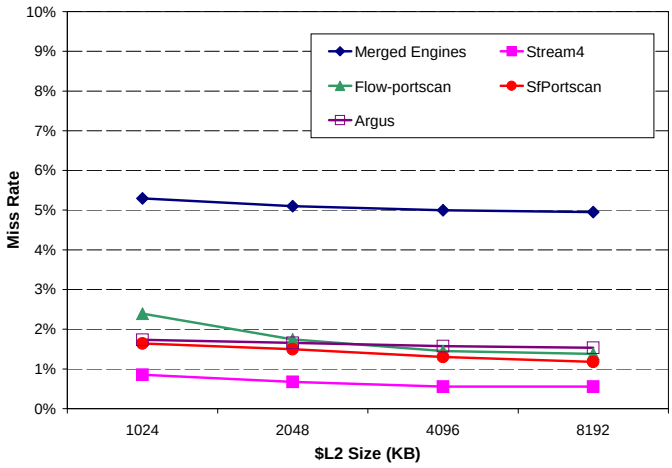
Figure 4.12(b) presents the L2 cache miss rate. There is a flat miss rate even with a large L2 cache. Nearly 5% of the memory accesses of Merged Engines are L2 cache misses, regardless the L2 cache size. Analyzing the misses, they are related to stateful data. Thus, under high traffic aggregation we cannot reduce the miss rate due to low temporal locality of flow state data even using several MBytes of cache. The network traffic properties have stronger influence on the performance of cache when the flow states are larger as well as there is more computational stateful workload.

Another interesting issue to discuss is the significant higher miss rate of Merged Engines versus other stateful configurations. There is no direct relation between higher miss rate and the combination of state data structures from a set of stateful engines. The key insight is the interaction of each stateful engine to trigger a new different application behaviour.

In order to go deeper in the analysis of stateful impact on cache, we analyze the L2 miss rate per packet. Figure 4.13 shows the L2 miss rate generated by the stateful data accesses of every packet through the flow lifetime. The process of the first packet of a connection shows the highest miss rate due to the initialization of the flow state. However, when the application is focused on detecting the safety of a TCP connection, such as Merged Engines, most of the misses are concentrated in the first packets, since most of decisions/actions are triggered in the first packets. In contrast, the applications with constant workload, such as Argus and Stream4, shows a similar miss rate in every packet, since the packet processing flow path is very similar among packets.



(a) $DL1 Miss Rate



(b) $L2 Miss Rate

Figure 4.12: Data Cache Behavior

4.5.3 Branch Prediction

We identify two different categories of branches within evaluated stateful benchmarks: flow dependent and flow independent branches. The former includes the branches that update the flow state and variables related to the flow, such as state of the connection. As the prediction is dependent on the flow, these branches are sensitive to network

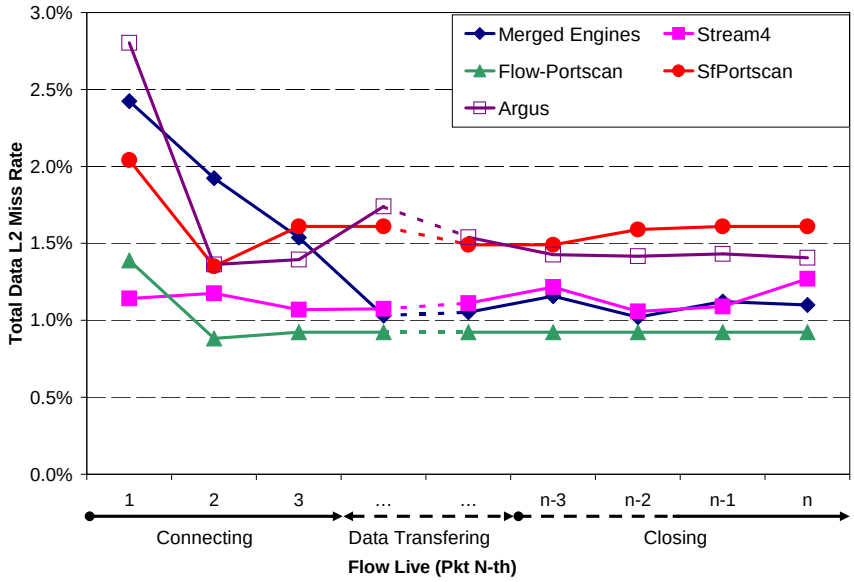
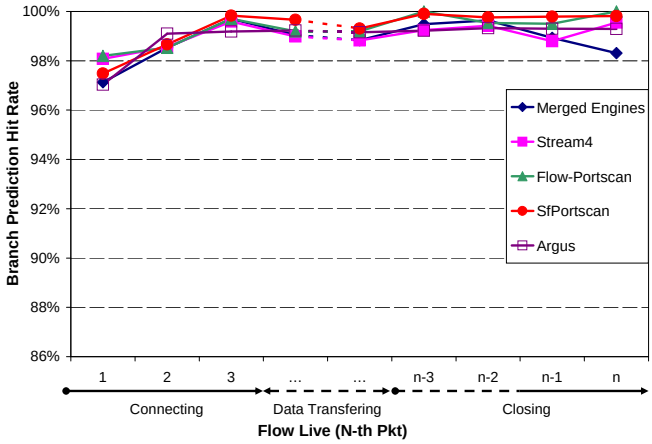


Figure 4.13: L2 Cache Misses per packet during the flow lifetime

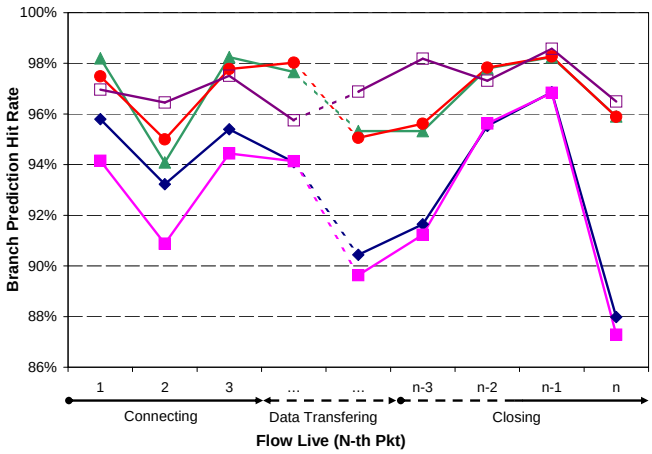
traffic features. Moreover, they are harder to predict due to a possible negative aliasing among packet processing. The latter includes the branches that are executed for packet data itself and global tasks, such as packet header values and global variables. These branches are insensitive to the previously mentioned traffic characteristics (see Section 3.2). Therefore, they are easy to predict, since the prediction is similar among packets. Other stateful applications with wider range of stateful levels show more categories of branches according to the association between state levels and branches.

Figure 4.14(a) presents the branch prediction analysis throughout the flow lifetime with an environment of a single active connection. We can observe all benchmarks present a high branch prediction hit rate in every packet. In fact, the hit rate is higher than the average (see Figure 4.8) and there are no significant variations among n-th packets. On the other hand, Figure 4.14(b) shows the results throughout the network connection lifetime with an environment of high aggregation traffic level. That is, there are many single packets of different unique flows leading to a polluted PHT. We can observe that, on average, there is a lower hit rate in every packet, specially Merged Engines and Stream4 present the most significant variations.

We can observe that there is a negative aliasing due to the processing of packets from



(a) Branch Prediction - Traffic without aggregation level



(b) Branch Prediction - Traffic with high aggregation level

Figure 4.14: Branch prediction study during the flow lifetime

independent flows. Due to this, there is a significant amount of flow dependent branches that are affected by the traffic characteristics. However, the majority of the evaluated benchmarks concentrates the main workload in the first packets of the connection. Thus, the negative impact on performance is lower, with the exception of Stream4 and Argus that present a similar workload throughout the n-th packets of the connection.

4.5.4 Performance Variations

In Section 4.5.1 we analyze the computational workload of every packet of a given flow. The performance metrics shown in Section 4.4.6 are an averaged of all processed packets. However, the results don't describe the performance of a particular packet. We use the baseline architectural configuration shown in Table 4.2. An out–of–order 4–width superscalar processor, with 64KB IL1$ and DL1$, and 2MB unified L2$.

In Figure 4.15 we break down the performance of every packet during the life of a given flow. The Y–axis denotes the IPC, while the X–axis indicates the n–th packet and the transport protocol state. The average IPC approximates to the global performance shown in Figure 4.9.

Mostly stateful applications show similar performance for every packet. After the first packet of a connection, there are slight increments of IPC, especially SfPortscan, Stream4, and Argus. In addition, Stream4 preprocessor shows higher performance in the middle of the connection. The reason behind this is that the required computation is lower, since the flow state is only checked and neither modifications nor further actions are triggered.

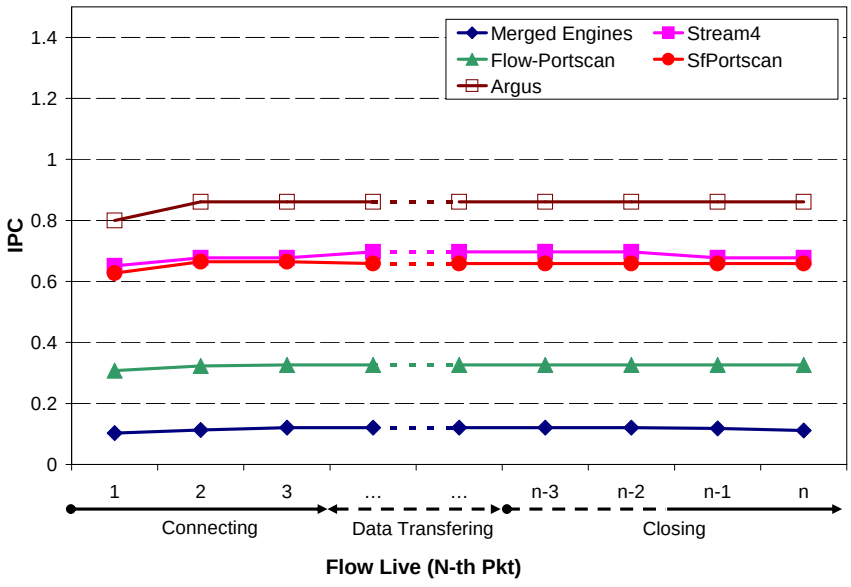


Figure 4.15: Performance per packet during the flow lifetime

In order to better analyze the performance variations, we present the normalized IPC in Figure 4.16. The IPC is normalized to the performance of the first packet of the connection. We can observe that Merged Engines manifests a curved trend. That is, the IPC is increased in the second and third packet up to nearly 1.17x. Then it is sustained until the last packets of the connection that presents lower performance. Merged Engines present higher differences than the other benchmarks, since it requires to do extra workload in the first packets that degrades the overall performance. In contrast, the other benchmarks demand less extra workload due to the flow establishment.

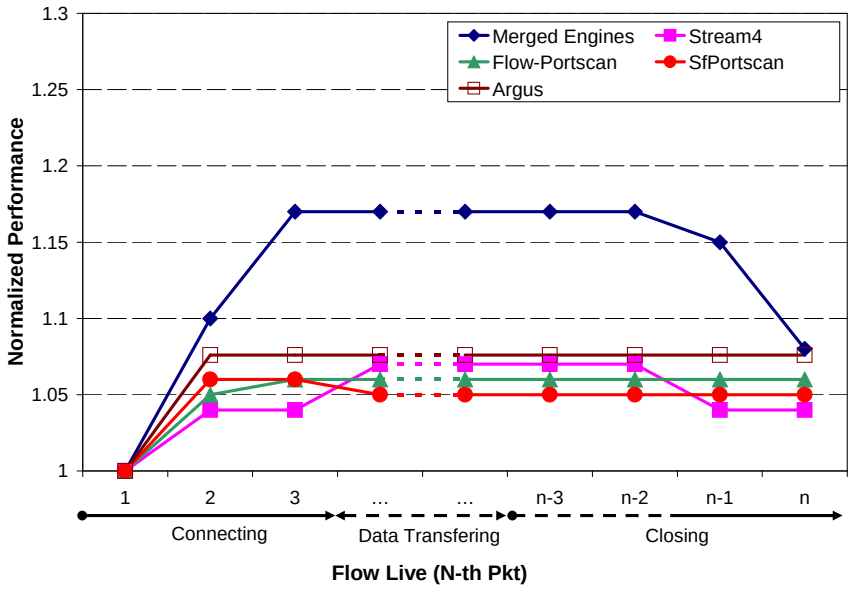


Figure 4.16: Variations of IPC per packet during the flow lifetime

Overall, the stateful benchmarks show a sustained increment of performance ranged from 4% to 8%, excepting Merged Engines. Applications that demand special processing for the first packets of the flow (*e.g.* security detection) increase the impact of extra computation on stateful data (performance penalty due to stateful data memory management). Even it may present slight variations during the flow lifetime (*e.g.* Stream4). Instead, regular packet processing (*e.g.* argus monitoring) shows stable performance for most of the packets.

4.6 Related Work

Benchmarking NPs is complicated by a variety of factors [13], such as confidential workload properties of proprietary applications and emerging applications that do not yet have standard definitions. There is a high interest and an ongoing effort in the NP community to define standard benchmarks [58].

Several benchmarks suites have been published in the NP area: CommBench [100], NetBench [54] and NpBench [44]. Wolf et al. [100] present the CommBench benchmark suite: a set of eight benchmarks classified in Header Processing Applications (HPA) and Payload Processing Applications (PPA). The suite is focused on program kernels typical of traditional routers. The workloads are characterized and compared versus SPEC benchmarks.

Memik et al. [54] present a set of nine benchmarks, called NetBench. The authors categorize the benchmarks into three groups, according to the level of networking application: micro-level, IP-level and application-level. The workloads are compared versus MediaBench programs.

Lastly, Lee et al. [44] propose a new set of ten benchmarks, called NpBench. It is focused on both control and data plane processing. In this case, the benchmarks are categorized according to the functionality: traffic management and quality of service group (TQG), security and media processing group (SMG), and packet processing group (PPG). The study of the workloads is compared against the CommBench workloads.

All the above benchmarks are not stateful applications, since they do not keep track of the previous processed packets. In reality NetBench includes the only benchmark that presents stateful features. The NIDS called Snort [10], although it is not included in the original paper [54]. There are several publications that present studies about Snort [70, 40, 75]. However, the workload and the cache behavior of the stateful configuration have not been analyzed yet.

The processing behavior of low layer applications of NP benchmark suites are very similar among packets. Thus, there is no characterization of workloads throughout the lifetime of a flow. In contrast, stateful applications show significant variations in both code footprint and data working set. The conclusions are important to propose alternative schemes for distributing workload in multithreaded architectures.

4.7 Conclusions

In this chapter we present the first workload characterization of stateful processing. To the best of our knowledge, this study is the first to analyze stateful workloads, specially for networking applications.

In order to better understand the bottlenecks that the network applications generate, we propose a new type of classification that distinguishes workloads according to the management of data throughout packet processing: self–contained (each packet contains all data needed for processing), stateless (shared data structures that are used to check packet data and trigger an action), and stateful (keep track of previous packet processing in order to provide higher knowledge about the current packet processing).

The results show important differences between workload categories. The main reason for the variation in performance among workload categories relies on the data locality. The study shows that stateful processing is related to state data and therefore it is sensitive to traffic flow characteristics. In stateless applications, however, the locality is sensitive to lower layer network features (e.g. IP address distribution), as well as to the size of global information required by the packet processing.

In a single threaded system, the main bottleneck of stateful applications is the memory system, since even a larger L2 data cache is unable to maintain the state of active flows. The evaluated applications show from 3x up to 19x of performance speedup on average using a perfect memory system. The speedup is sensitive to the statefulness degree of the application and to the memory hierarchy configuration of the processor architecture. Nevertheless, branch prediction is another critical issue once memory bottleneck is overcomed, showing 16% on average of speedup. Also the percentage is sensitive to the application characteristics and the state level processed during the packet processing.

In this chapter we also analyze the processing for each packet during the life of a network connection. We compare packet requirements under different stateful configurations of Snort. The analysis shows that the bottlenecks can be concentrated in the first packets of the TCP connection, such as portscan detector, or can be distributed along the flow lifetime, such as monitoring.

The variations increase as more stateful upper layer processing is performed to the network traffic. Although this chapter discusses the variations among packets of a given TCP connection, the performance can also be sensitive to the state related to other network layers, such as group of flows, traffic of a given user, traffic of a given application. Other applications may present different valuable results, but our studies suggest that the

critical bottlenecks will be preserved and even be more stressed.

The analysis is performed in single threaded processor. In Chapter 6 we analyze bottlenecks that arise due to parallel execution models in a multithreaded architecture (*e.g.* negative aliasing in both data and instruction caches, contention due to shared data structures).

Chapter 5

Principles of Parallel Stateful Processing

In previous chapters we have characterized the stateful applications. We have demonstrated that stateful applications present computational intensive workloads, reduced ILP, and large amount of long latency memory accesses. Thus, providing more complex stateful DPI services for high bandwidth links leads us to explore multithreaded architectures to exploit other levels of parallelism, such as the inherent packet level parallelism of network processing.

There are two main execution models towards parallel network processing on multithreaded architectures: run–to–completion (RTC) and software pipeline (SPL). In the RTC model, a thread is run to process packets in a discrete, indivisible RTC step. In the SPL model, a thread is run to process packets in several steps according to the differentiated pipeline stages of the workload. Both models aim to exploit packet level parallelism (PLP). In addition, SPL aims to improve the affinity among threads.

Layer 2–3 processing exploit large amount of PLP, since there are negligible dependencies among packets. In contrast, stateful processing shows significantly lower PLP, since there are dependencies between packets (*e.g.* variety of states related to a particular packet) as well as contention in the use of shared data structures.

In this chapter we introduce background information of parallel stateful processing. The analysis is based on our parallelization of Snort [70], called Snort–MT. We present

an analysis of the parallel workload. In addition, we introduce both RTC and SPL execution models using Snort–MT.

5.1 Motivation

In this section we present an example of throughput demand to provide Snort stateful DPI while sustaining 10Gbps traffic (*i.e.* OC–192 at full bandwidth utilization). We assume a packet size of 500 bytes on average [53] and workloads from different Snort configurations (see Table 6.1). In addition, we assume a massive multithreaded architecture where each stream is single–issue and the processor runs at 1GHz. Figure 5.1 depicts the required amount of streams in the architecture (Y–axis) running different Snort workloads (X–axis).

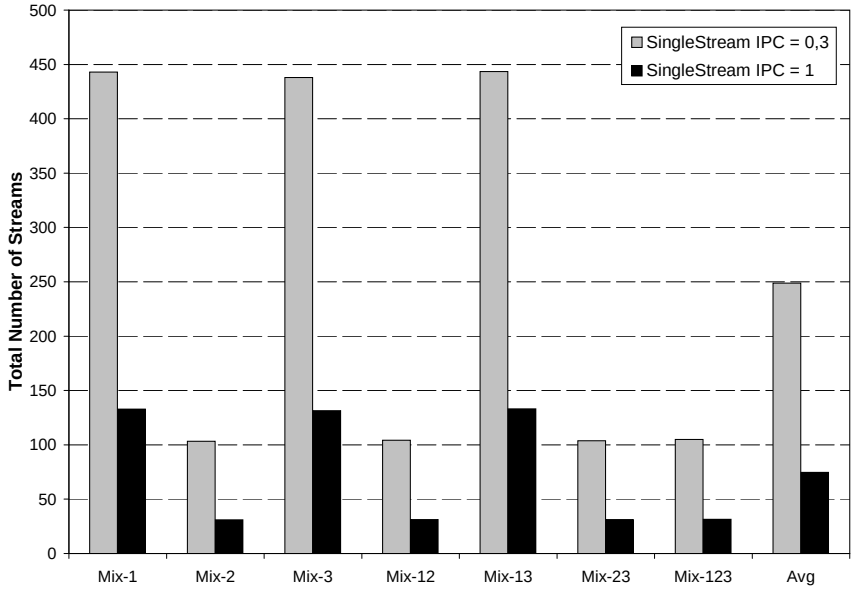


Figure 5.1: Massive multithreaded size to sustain stateful DPI for a 10Gbps link

The black bars indicate the architecture size assuming the best stream performance case (*i.e.* IPC of 1 per stream). We can observe that the total number of streams ranges from 30 to 130. However, as we demonstrated in Chapter 4, the IPC of stateful DPI applications is substantially lower. The gray bars indicate the requirements assuming a

stream performance 0.3 IPC. It is the average performance of single threaded superscalar processor shown in Section 4.5.4. In this case, we need from 100 to about 450 streams in the processor to sustain required throughput.

In fact, the theoretical requirements indicated in Figure 5.1 increase in a real environment. The dependencies among threads reduce the linear scalability of performance, especially for architectures with tens to hundreds of streams.

5.2 Workload Breakdown

The parallel network applications assign packets to software contexts (i.e. threads). The majority of network systems use the same amount of threads than the total number of hardware contexts (i.e. streams). In this section we discuss the workload of a given thread assigned to a particular stream.

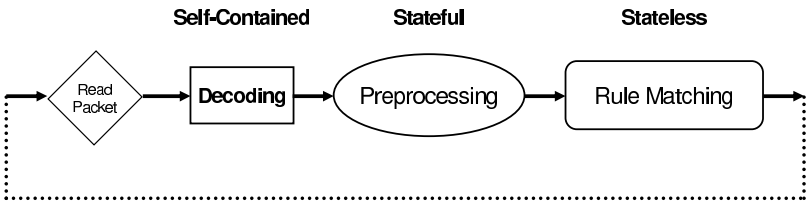


Figure 5.2: Snort packet processing loop

Figure 5.2 depicts the packet processing loop of Snort. According to the workload classification proposed in Section 4.2 (i.e. self–contained, stateless, stateful), there are three main stages well differentiated: decoding, preprocessing, and rule matching. Once the stream finishes the packet processing, the system releases the thread and checks if there is any packet waiting to be processed (dotted line). Figure 5.3 presents the multithreaded Snort design that we develop in our implementation. We replicate the packet processing loop per thread and there is no breaking of the packet processing loop.

The received packet from the network card (i.e. packet capturing) is allocated into the memory waiting to be processed. The system assigns the packet to a given thread that is processed in a particular stream. The processing starts to decode the packet header and payload to initialize a number of data structures (i.e. decoding stage). The decoding

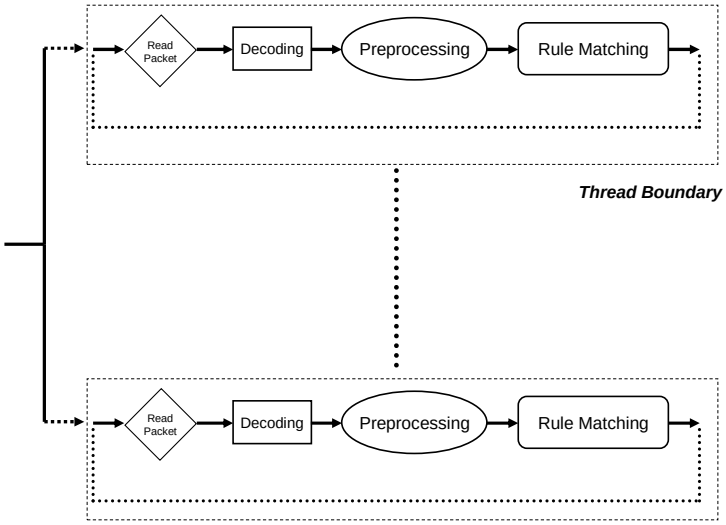


Figure 5.3: Multithreaded Snort packet processing

stage presents self–contained workload, since it only needs data from the packet itself. The preprocessing engine receives the packet information and perform specialized tasks. They can behave as either stateless or stateful processing depending on the configuration. Nevertheless, throughout the experiments of this thesis we only enable stateful preprocessors in order to enhance the stateful processing. The stateful preprocessing keeps track of previous processed packets. Finally, the packet is scrutinized for signature matching. This stage is categorized as stateless workload, since external information is required (*e.g.* rules, signatures, keywords) and there is no need to keep track of previous packet processing.

The description of Snort packet processing points out that stateful DPI comprises several workload categories. This assumption can be extended to other stateful DPI applications, but with different distribution rates. In contrast, layer 2–3 applications present a single workload category (*e.g.* IP forwarding presents stateless workload).

We base our studies on different configurations of the Snort as a representative range of stateful DPI applications. We select a number of stateful preprocessors that show different behavior:

- *perfmonitor:* it collects a wide range of statistics from packet processing intended for network administrators.
- *stream4:* provides TCP stream reassembly and stateful analysis capabilities to track simultaneous TCP streams and to ignore stateless attacks.
- *frag3:* is an IP defragmentation module that applies target-based host modeling anti–evasion techniques for attacks based on information about how an individual target IP stack operates.

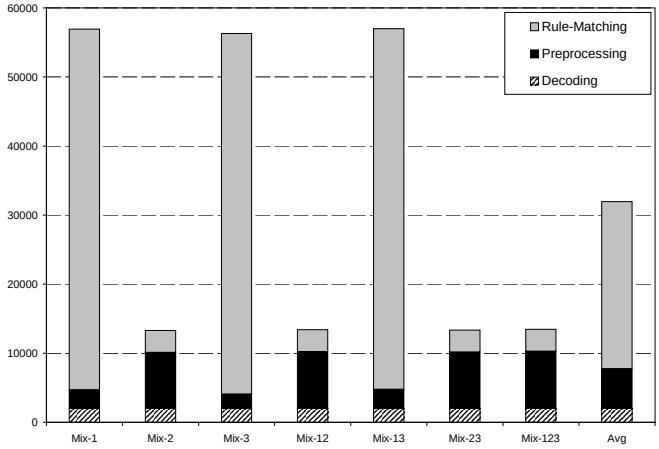
Each workload presents a different configuration of enabled preprocessors. Table 5.1 indicates the enabled preprocessors according to the workload identificator. Moreover, in all configurations we employ the default configuration of rule set for rule–matching, that includes a total of 3291 rules.

Enabled Preprocessors	Workload ID
Perfmonitor	Mix-1
Stream4	Mix-2
Frag3	Mix-3
Perfmonitor - Stream4	Mix-12
Perfmonitor - Frag3	Mix-13
Stream4 - Frag3	Mix-23
Perfmonitor - Stream4 - Frag3	Mix-123

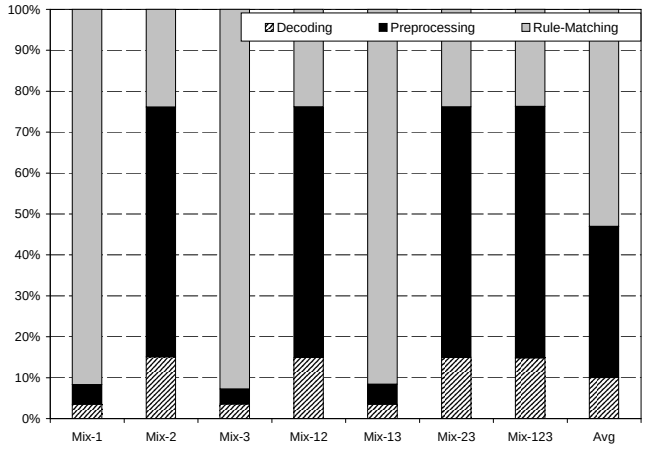
Table 5.1: Workload Mixes

Figure 5.4 shows the Snort workload distribution per packet processing stage. The X–axis indicates the stateful DPI workload mix according to the configurations of Snort. They differ in the enabled preprocessors. In the top graph we can observe the number of instructions denoted by Y–axis. The decoding stage (the bars with diagonal lines) shows similar workload (about 2K instructions per packet) regardless the Snort configuration, since there are no directives for setting up the Snort decoder. In contrast, the preprocessing workload (indicated by the black bars) presents variations ranged from 2K up to about 8.3K instructions per packet, due to the different preprocessor configurations. Regarding the rule–matching stage (the gray bars), there are significant workload variations, although all configurations preserve the same rule–set. The reason behind this is that the preprocessing provides further knowledge to the packet processing and it can skip subsequent preprocessing or rule–matching for a given packet. We can observe the configurations that show reduced preprocessing workload (*i.e.* Mix–1, Mix–3,

Mix–13) present higher rule–matching workload than the rest of configurations. Thus, rule–matching workload is sensitive to the preprocessing configuration.



(a) Instructions per Packet



(b) Workload Distribution

Figure 5.4: Snort workload distribution according to the processing stages

In addition, enabling more preprocessors doesn't increase linearly the workload per packet. Part of the knowledge provided by a given preprocessor is used by other preprocessors (*e.g.* data structures). For example, the preprocessor workload of Mix–1, Mix–3, and Mix–13 or Mix–1, Mix–2, and Mix–12 show marginal differences since they share part of the processing workload. In addition, the remaining preprocessing

may present lightweight computational workload (*e.g.* Mix–1 updates a number of counters), unlike other preprocessors with more complex (*e.g.* Mix–2 provides TCP stream reassembly).

Figure 5.4(b) depicts the percentages for each processing stage. This graph clarifies the distribution of instructions per packet processing mentioned above. We can observe that there is not a direct relationship between the number of preprocessors enabled and the percentage of preprocessing and rule–matching workload. Most of the stateful DPI presents stateless and stateful workload. In fact, the configuration Mix–123 comprises the highest number of stateful preprocessors, whilst the packet processing presents about 60% of stateful workload.

It is very likely that more complex network services (*e.g.* complex network security and monitoring systems) will show future stateful DPI applications with larger computational workload requirements (*i.e.* number of instructions per packet processing) as well as stateful requirements (*i.e.* amount of state and stateful workload).

5.3 Critical Sections

The stateful applications present a large amount of critical sections to protect shared data structures. The frequency of data updating is high leading to significant thread collision rates in critical sections. Instead, lower layer applications comprise just a few critical sections. In fact, the self–contained packet processing don't share information among packet processing and thus there are no collisions between parallel in flight threads. In contrast, stateless workloads check data in global shared structures that show low updating frequency (e.g. update frequency of IP forwarding table), so the probability of thread stalling due to thread collision in a critical section is marginal.

In this thesis, we develop a suboptimal implementation of Snort version 2.3.4 [81], called Snort–MT. We employ Pthread POSIX standard[1]. Although the design of critical sections can be improved (e.g. finer grained locking and optimized allocation of locks), the nested critical sections cannot be avoided.

[1]In Section 7.2.2 we outline some future work that addresses alternative parallel implementations of Snort by using OpenMP [1] (*i.e.* a portable, threaded, shared-memory programming standard API with "light" syntax) and Transactional Memory [43] (*i.e.* a promising mechanism that simplifies parallel programming by providing a new abstraction to the protection of shared data).

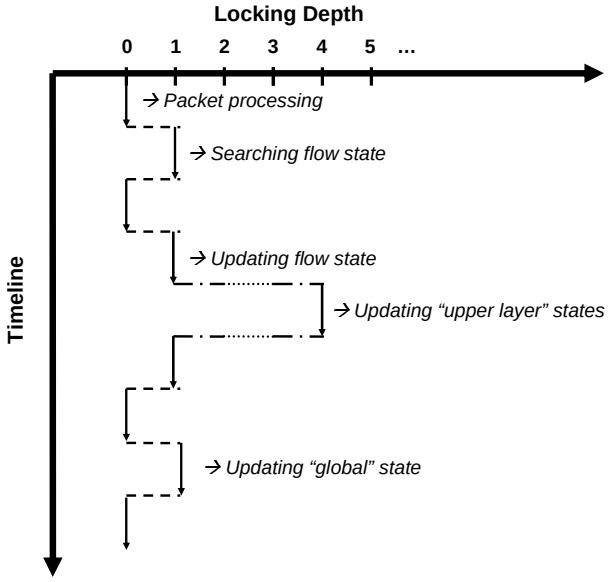


Figure 5.5: Example of nested locking due to stateful processing

Figure 5.5 depicts an example nested locking scenario due to stateful DPI. The example is based on Snort stateful preprocessing, but it can be extended to other stateful network applications. The X–axis denotes the nesting depth level and the Y–axis indicates timeline. The lock free processing (*i.e.* zero locking depth) is related to the computation of packet contents. The first level of locking is either related to the data structure of active flow states or global statistics. Processing a particular flow state requires to check key data and eventually to update or to trigger an action. Other upper layer states (e.g. user stats) may need to be updated during the period of flow state processing.

We can observe in Figure 5.6 the distribution of critical sections and nested levels of locked code in Snort–MT. We assume a particular stateful configuration of Snort–MT (see Workload Mix–123 in Table 6.1). The processing workload presents nearly 30% of the processing within critical sections dinamically distributed across 10 critical sections on average. In fact, the rule–matching stage shows negligible critical sections (less than 1%). The decoding stage presents about 12% of locked sections, due to global counters (*e.g.* number of TCP packets). Most of them present a size of 100 instructions on average and a frequency that ranges from about 500 to 1K instructions. The decoding stage usually executes 3 critical sections on average.

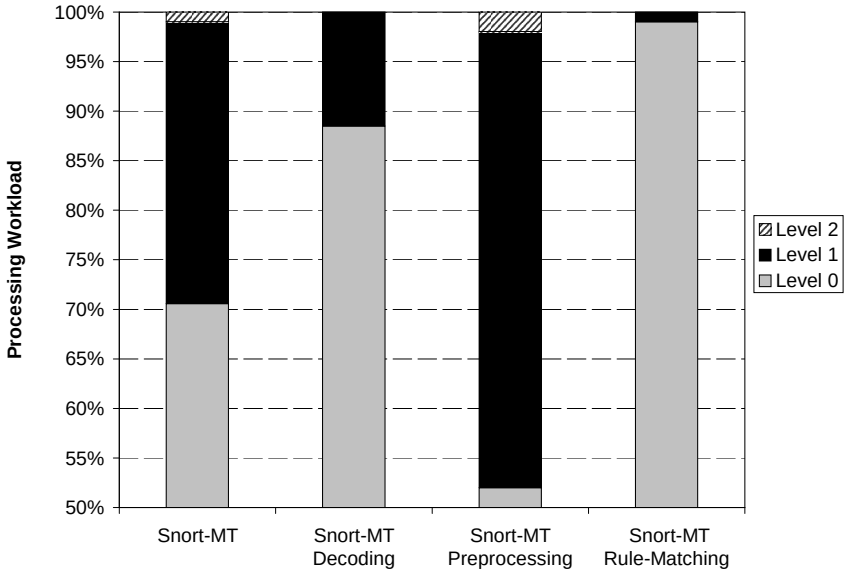


Figure 5.6: Distribution of critical sections and nested levels

In contrast, the preprocessing stage (*i.e.* mostly stateful workload) shows higher percentage of locked processing, about 48%. In addition, 2% of the critical sections present nested locking (see example in Figure 5.5). In fact, applications that keeps larger state categories (*e.g.* flow–state, application–state, user–state) will likely present deeper nested levels. During the preprocessing, there are nearly 6 critical sections executed. About 30% of the critical sections present a size that ranges from 2K to 3K instructions (*e.g.* management of flow states), while the rest of critical sections shows a size of over 100 instructions (*e.g.* updating a particular state). The distance among critical sections ranges from 200 to nearly 2K instructions.

5.4 Order–of–Seniority

The stateful processing presents critical sections that require to be processed following a particular order. Most critical sections that need order enforcement are based on the order of packet arrival, also known as order–of–seniority (see Section 2.3). Thus, a given thread can be stalled in a critical section, even there are no collision with other threads

at a given time.

The probability of thread stalling due to order–of–seniority is sensitive to two factors: the temporal locality among packets that use a given state, and the amount of processing that require order enforcement. For example, keeping network statistics need no order–of–seniority, unlike keeping the state of TCP handshake protocol.

Our experiments running Snort under high traffic aggregation present no conflicts due to order–of–seniority. There are stateful data structures that require order enforcement, but the high traffic aggregation leads to reduced temporal locality. The minimum distance between two packets that requires the same state data present show hundreds of packets. Further work is needed with other benchmarks to evaluate the effect of order–of–seniority as part of the future work in Section 7.2.

5.5 Parallel Execution Models

There is a relationship between the execution model and the approach to map parallel workloads into a given multithreaded architecture. The parallel execution models adapt workloads to enhance the exploitation of parallelism. The mapping approach takes advantage of the execution model benefits and distribute the parallel workload to enhance the positive interaction among threads and thus to maximize the system throughput by exploiting packet level parallelism.

In this section we introduce the two main execution models for parallel network applications: Run–To–Completion, where a packet is processed in a single step; and Software Pipeline, where the packet processing comprises a number of pipelined stages.

5.5.1 Run–to–Completion

The Run–To–Completion (RTC) execution model is the simplest approach to parallelize a workload. The packet is processed in an indivisible single step. The processing loop is entirely replicated for every thread. As we mentioned in Section 5.2, there is the same amount of threads than the total number of streams. Figure 5.7 depicts the generic RTC model of Snort–MT assuming a multicore multithreaded architecture. We can observe all streams run the entire Snort processing loop assigned to a thread.

The RTC model doesn't specialize cores by gathering threads with similar execution

footprint or data working set. In addition, unless the OS migrates a given thread from a core to a remote one, the RTC model shows no thread migrations.

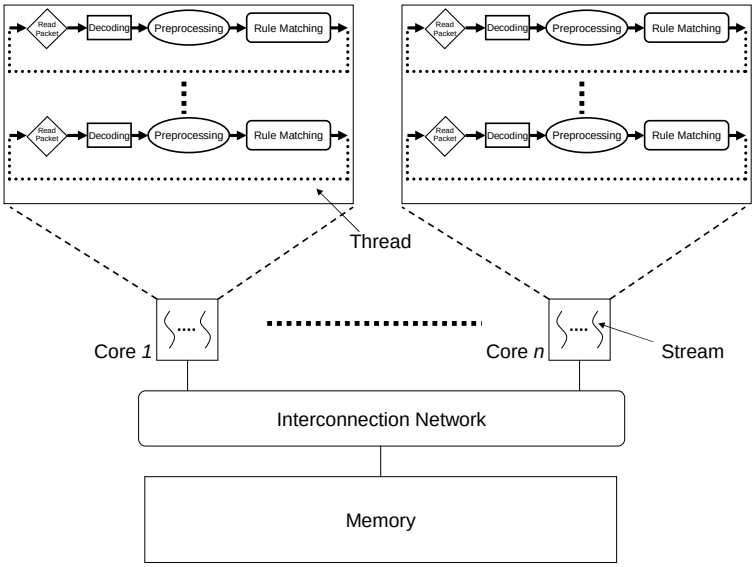


Figure 5.7: Run–To–Completion Execution Model

The RTC model is suitable for short workloads with similar execution footprint among packet processing, such as layer 2–3 applications. Such workloads show reduced instruction footprints as well as reduced data working sets. Thus, increasing the number of threads running in parallel per core improves the memory performance by enhancing positive aliasing in both instruction and data caches. As those applications do not update shared data structures, there is no interaction with threads executed in remote cores.

The stateful DPI applications show large execution footprint and large data working set with significant variations among packets. For this reason, it is very likely to increase negative affinity between threads while scaling the number of streams per core. In addition, there are dependencies between packets (*e.g.* flow state, global counters) that can reduce the scalability of this methodology.

5.5.2 Software Pipeline

In the Software–Pipeline (SPL) execution model the packet processing loop comprises a number of steps, called pipeline stages. Threads executing the same stage ideally are assigned to streams in the same core. The SPL model aims to specialize the instruction and data cache of cores by gathering threads from a given pipeline stage. The code footprint and the data working set of a given pipeline stage should be similar for most of the packets and should present minimum dependencies with other cores. The main goal of SPL is to overcome the negative cache affinity among threads under the RTC model. Thus, increasing the number of threads assigned per core preserves the improvement in the memory performance due to positive aliasing among threads on both instruction and data caches.

One of the main proposals in the literature that addresses the SPL model is the methodology presented by Weng et al. [98]. The authors use an annotated directed acyclic graph (ADAG) to represent clusters of instructions. Each node provides information about data and control dependencies between instructions. The goal is to minimize inter–cluster dependencies, while maximizing cluster size of cohesive nodes. However, the algorithm that generates the graph doesn't take into account any information about dependencies with other packets, states, or even dependencies due to shared data structures.

There are two different approaches to fit an application into a SPL model. On one hand, the compiler automatically generates the pipeline stages according to a given set of heuristics [98]. This distribution is statically generated at compile time. On the other hand, the application developer manually introduces the boundaries of each pipeline stage and deals with the communication mechanism between stages.

We can observe in Figure 5.8 a generic SPL model of Snort–MT assuming a multicore multithreaded architecture. We can observe three pipeline stages: decoding, preprocessing, and rule–matching. Each core executes a set of threads related to a given pipeline stage. If there are more cores than pipeline stages, multiple cores will be assigned to each particular stage. There is a mechanism to communicate from a particular pipeline stage to the next one. The graph depicts ”pipeline buffers” as the ”latches” for SPL communication between pipeline stages.

There is a variety of mechanisms to communicate pipeline stages: thread migration, where the context of a thread is moved to a remote core (*e.g.* Flowstorm processor [53]); special purpose register set file, where the stream accesses to the registers of a stream placed in a remote core (*e.g.* Tilera64 [8], IXP family processors [34]); and memory

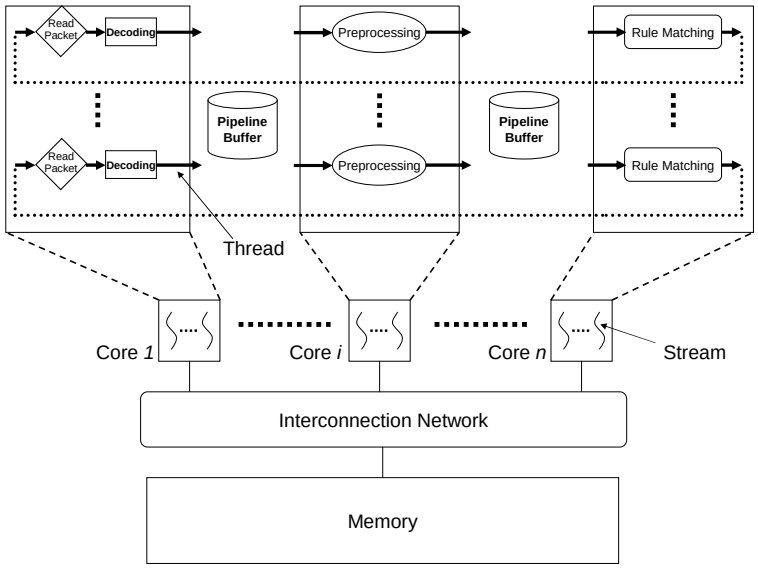


Figure 5.8: Software Pipeline Execution Model

buffer queues, where pointers to the required data structures are moved to buffers in memory (*e.g.* UltraSPARC T1&T2 [86, 26]). The selection of the proper mechanism is based on the characteristics of both the application and the processor architecture.

The SPL model is suitable for workloads related to layer 3 network processing. That is, applications that present larger workloads than layer 2 network processing [53] (*i.e.* hundreds of instructions per packet) with no dependencies among packets processed in parallel. In addition, the applications comprise well differentiated pipeline stages. Threads of a given pipeline stage positively interacts in instruction and data cache. Although there are shared data structures used by different packet processing, there is negligible contention on critical sections. Most of the times shared data structures are accessed to check a particular value, but they are occasionally updated.

However, stateful DPI applications present a variety of obstacles that leads to theoretically reduce the benefits of the SPL execution model. Stateful DPI present irregular code footprint and significant variations in data working set among packets. Especially the stateful applications, since the processing deals with a variety of states related to the packet that can trash with the data required by other packets. In addition, other shared data structures are used in different moments of the processing timeline. Thus,

the workload shows fuzzy pipeline stages and it is likely to arise dependencies among remote pipeline stages. For these reasons, it is difficult to fit a stateful DPI application into a SPL execution model. Especially the near future stateful DPI applications will enhance this problem leading to significantly reduce the benefits of the SPL approach.

Chapter 6

MultiLayer Processing

This chapter presents the MultiLayer Processing (MLP). MLP is an execution model for exploiting several levels of parallelism through the specialization of core sets. The methodology categorizes different stateful data structures. Then, it differentiates sections of code according to the related stateful data category, called processing layers. We distribute the processing resources of a massive multithreaded architecture according to the processing layers. The results show that MLP is able to increase the positive aliasing and reduce negative aliasing between threads in both data and instruction cache. In addition, MLP alleviates contention in critical sections in heavy parallel systems. Throughout this chapter we analyze the benefits and limitations of this methodology and compare MLP against other parallel execution methodologies.

6.1 Chapter Roadmap

This chapter is outlined as follows. Firstly, we describe some critical features of parallel stateful processing that lead to categorize the variety of stateful data categories. Moreover, we introduce the ideal parallel stateful processing.

Section 6.2.2 describes the MultiLayer paradigm. We define the concept of processing layer and explain how the MultiLayer Processing (MLP) works. Subsequently, Section 6.2.3 introduces the identification of processing layers and an example of execution. In Section 6.4 we present the evaluation of the MLP and other execution models. Finally,

we conclude with the survey of some related work and presenting the conclussions of this chapter.

6.2 MultiLayer Processing

RTC and SPL execution models present several issues that cannot optimally overcome. On one hand, although RTC model is the simplest approach it creates negative aliasing among threads assigned to streams of a given core. The whole packet processing is executed in a single core leading to trashing data and instructions among streams. On the other hand, SPL model differentiates packet processing pipeline stages and thus it reduces negative aliasing rates. While this benefits are shown especially in low layer network applications, stateful upper layer applications don't present such improvement. The reason behind this is that stateful data structures are spreaded over all the packet processing. The pipeline stages are not capable of isolating well differentiated sections of code. Moreover, there are large variations of flow paths in a particular pipeline stage due to the computational complexity of upper layer packet processing. Limitations of RTC and SPL are emphasized in massive multithreaded architectures that present reduced caches per processing node for scalability reasons.

We identify the following key features of parallel stateful applications:

- The applications present sections of code related to different stateful data structures. They can be classified into a variety of categories called processing layers.
- The classification of processing layers offers multiple levels of parallelism.
- Shared data structures are processed several times during the packet processing lifetime. Therefore, it is difficult to bind a given shared data structure to a pipeline stage as the SPL execution model does.

Throughout this section we describe a proposal to take advantage of the characteristics mentioned above in order to maximize throughput of parallel stateful processing. First, we describe the basics of the ideal parallel stateful processing. Then, we define the MultiLayer paradigm as well as the methodology to modify the code. Finally, we describe a generic implementation of the mechanism and discuss a variety of tradeoffs.

6.2.1 The Ideal Parallel Stateful Processing

The main goal of a parallel stateful processing is to maximize the throughput system by exploiting as many levels of parallelism as possible. To do this, threads must be assigned to streams in such a way that dependencies among them are minimized, while resource utilization provides the maximum performance of the system. In addition, the paradigm must reduce conflicts in critical sections, reduce negative cache affinity among streams, and maximize the reusability of data and instructions among different packets, such as stateful data.

In Chapter 4 we demonstrate that flow state data present very low temporal locality under high traffic aggregation links. Thus, even large caches are not capable of keeping a particular stateful data. However, not all the stateful data present the same temporal locality.

6.2.2 MultiLayer Paradigm

6.2.2.1 Processing Layers

We define processing layer as a section of code related to the processing of a particular stateful data. In the network area, there is a wide variety of stateful data categories as we introduce in Section 4.2.

We classify the stateful data into two groups: a) the main stateful categories (directly related to particular layers of the TCP/IP network stack), and b) the stateful sub–categories (that differentiate multiple stateful data groups within a given main category). On one hand, the classification denotes hierarchical categories. On the other hand, each category can be processed in parallel with others as far as there are no dependencies among them. Thus, processing layers related to those state categories expose multiple levels of parallelism.

In the Snort–MT benchmark we identify three main categories: packet, transport, and global layers. Then, within transport layer, we sub–categorize three additional layers: Flow, TCP Flow, and Fragmented Packet layers. In addition, global layers can be also sub–categorized in two additional layers: Stats and Rule layers. Thus, according to our manual classification of the Snort–MT code, there are six stateful data categories, as shown in Figure 6.1 with the gray boxes. Some sub–categories may have dependencies among other sub–categories within its parent category. In our scenario, TCP Flow and

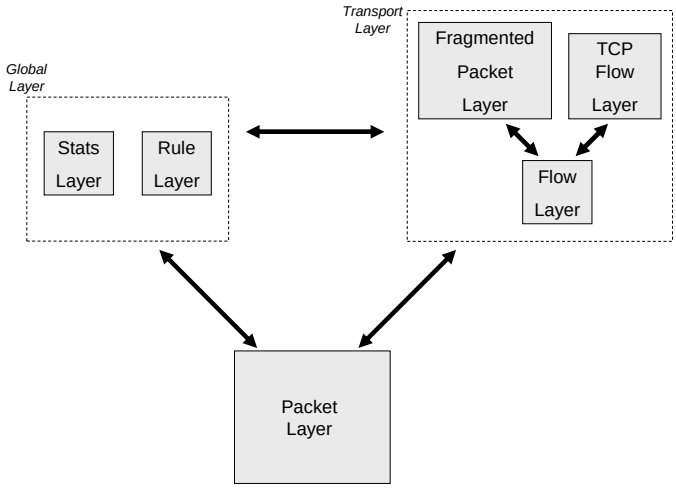


Figure 6.1: Stateful data classification of Snort–MT

Fragmented Packet layers are connected to the Flow layer, since there are shared stateful data structures.

In Section 6.2.3 we describe the approach to identify the processing layers (sections of code) related to every stateful data category.

6.2.2.2 MultiLayer Processing

We define MultiLayer Processing (MLP) as a paradigm to exploit the parallelism of workloads that comprise multiple processing layers. The basic idea is to exploit each processing layer by assigning groups of streams (*e.g.* multistreamed cores) to a particular layer. In a multistreamed multicore architecture, the OS initializes the system by assigning the cores to the processing layers [50].

Other techniques focused on exploiting multiple levels of parallelism [7, 57] identifies the maximum parallelism contained in the application through data and control dependence analysis. However, MLP aims at classifying sections of code using the network based knowledge of the stateful data classification.

Figure 6.2 shows the MLP execution model of Snort–MT. The system assigns a

packet to a particular available thread and this is assigned to a given stream. Once the thread needs to move from the current processing layer to another, it migrates to the proper core assigned to the required processing layer. Unlike the SPL model (see Figure 5.8), a given processing layer can be accessed several times during a single packet processing. For example, the flow state is accessed at the beginning and the ending of a packet processing.

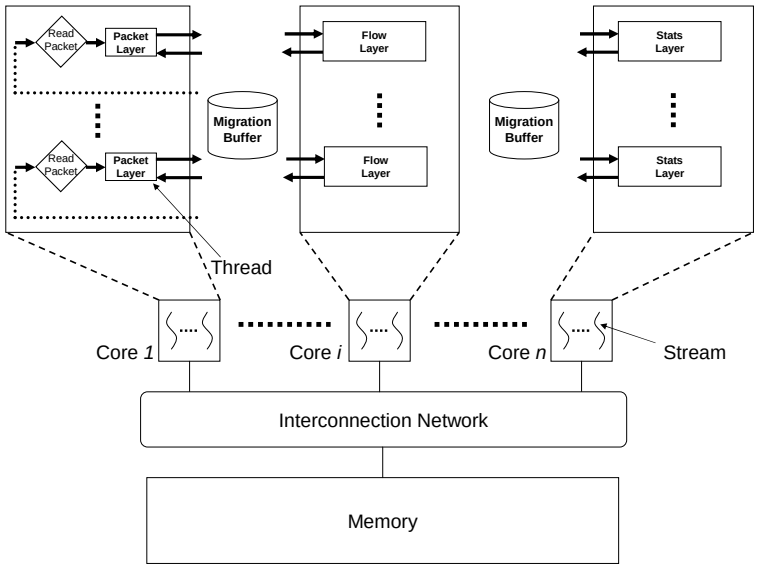


Figure 6.2: MLP execution model of Snort–MT

Threads migrate at the request of an explicit instruction introduced by the programmer. It is quite straightforward for the programmer to identify main stateful processing layers. However, it is more difficult to optimally differentiate sub–categories of such layers. The programmer provides the total number of application' processing layers. This control information is included in the executable at compile time. Thus, the programmer doesn't need to know the underlying architecture. Once the application is launched, the OS uses such information to initially assign cores to processing layers. A similar approach is employed for distributing multiple cores to use several virtual machines [50]. If there are more processing layers than cores, negative aliasing can be experienced as we show in Section 6.4.

6.2.3 Identification of Processing Layers

We use a recursive approach to identify processing layers. We analyze the parallel code of Snort–MT and identify the main stateful data categories. That is, those categories that can be directly related to different network layers according to the TCP/IP stack. Every section of code related to a particular stateful data category denotes a processing layer.

```
Processing(Pkt)
{
  ...
  TCP_Pkt_Counter++;
  ...
  Flow = Flow_Search(Pkt_addresses);
  if (Monitoring_Protocol(Pkt))
    Update_User_Stats(Pkt, Flow);
  else
    Update_Flow_Stats(Flow);
  ...
  if (Safe(Flow)) then
    Action_Safe(Flow);
  else
    Action_Not_Safe(Flow);
  ...
}
```

```
Update_User_Stats(Pkt, Flow)
{
  ...
  Pkts_User_Counter++;
  ...
  User = User_Search(Flow);
  ...
  if (Features(Flow) == Allowed_Flow(User))
    Update_Flow_User(Flow, User);
  else
    Action_NotAllowed(Flow, User);
  ...
  Update_Global_Counters(User,Flow);
  ...
  Update_Application_Data(Flow);
  ...
}
```

Figure 6.3: Example of stateful packet processing code

We assign sections of code to each category. We also perform a graph representation of the analyzed code integrating the section code of a particular processing layer into a single node. The next iteration aims at sub–categorizing different stateful data layers within a given node. We distinguish three possible scenarios for a particular node: there are multiple sub–categories with no data dependencies; there are multiple sub–categories that present marginal data dependencies; and there are multiple sub–categories that present significant data dependencies among them.

A conservative MLP approach doesn't split a given node that presents significant data dependencies among layer sub–categories. An eagger approach sub–classifies a given layer until there are no additional sub–categories within a particular layer category. The eagger implementation exposes larger variety of processing layers and then exloits MLP more than a conservative approach. However, the eagger design can experience data dependencies among processing layers, whereas the conservative implementation show marginal dependencies among layers. Our MLP development of Snort–MT assumes a conservative implementation.

Figure 6.3 shows an example of source code of a stateful packet application. The

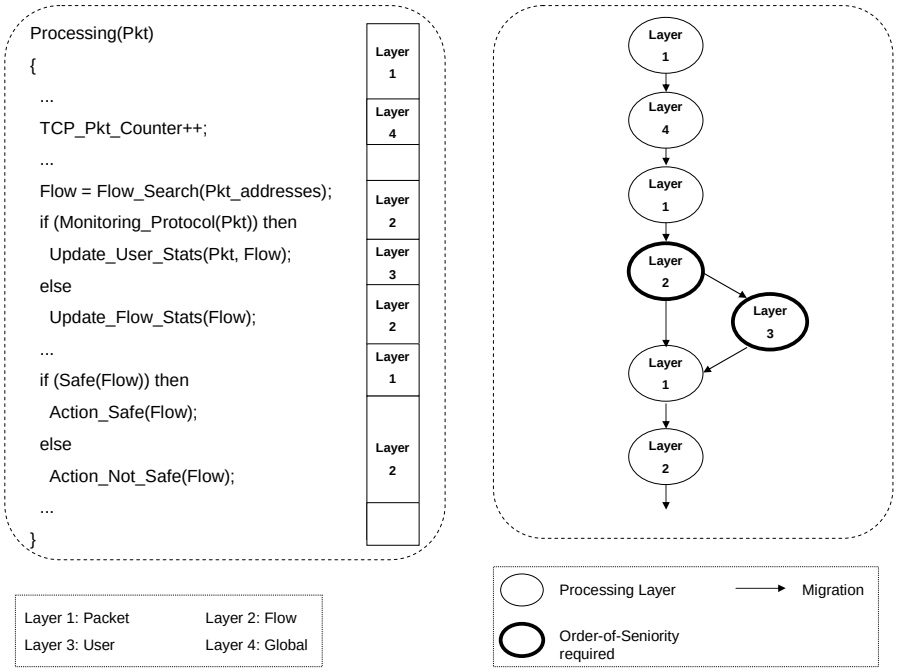


Figure 6.4: Intermediate representation of Function ”Processing”

example is based in one of the preprocessors of Snort (i.e. Stream4), but extended with features of stateful applications that keeps other state categories. We remark that all the sections of code that includes the management of stateful variables must be protected by a synchronization mechanism (*e.g.* lock–unlock, transactions). The update of a given stateful structure can lead to the modification of other stateful structures. The left column indicates the section of code used to process a packet. Firstly, some global stats are updated. Then, the flow related to the current packet is searched according to the packet header. If this is a new flow, then the state is inserted and initializated in the global flow data structure. Otherwise, the flow stats are updated. In this example, we also deal with user stateful data (the right column of Figure 6.3).

In Figure 6.4 we can observe the intermediate representation of the function ”Processing”. The left side of the figure depicts the relationship between the sections of code and the processing layers, according to the stateful data classification of the bottom–left of the figure. On the right side of the graph we can observe the migration flow path (denoted by the arrows) between a processing layer and others. We highlight

two nodes in order to show specific sections of code that can require order–of–seniority processing.

```
Processing(Pkt)
{
1 MIGRATE(Global_Layer);
   TCP_Pkt_Counter++;
2 MIGRATE(Packet_Layer);
  ...
3 MIGRATE(Flow_Layer);
  Flow = Flow_Search(Pkt_addresses);
  if (Monitoring_Protocol(Pkt))
  {
4   MIGRATE(User_Layer);
    Update_User_Stats(Pkt, Flow);
  }
  else
    Update_Flow_Stats(Flow);
5 MIGRATE(Packet_Layer);
  ...
6 MIGRATE(Flow_Layer);
   if (Safe(Flow)) then
     Action_Safe(Flow);
   else
     Action_Not_Safe(Flow);
7 MIGRATE(Packet_Layer);
  ...
}
```

```
Update_User_Stats(Pkt, Flow)
{
8  MIGRATE(Global_Layer);
    Pkts_User_Counter++;
9  MIGRATE(User_Layer);
   ...
10 MIGRATE(Flows_User_Layer);
    User = User_Search(Flow);
    if(Features(Flow) == Allowed_Flow(User))
      Update_Flow_User(Flow, User);
    else
      Action_NotAllowed(Flow, User);
11 MIGRATE(Global_Layer);
    Update_Global_Counters(User,Flow);
12 MIGRATE(User_Layer);
   ...
13 MIGRATE(Application_Layer);
    Update_Application_Data(Flow);
14 MIGRATE(User_Layer);
   ...
}
```

Figure 6.5: Example of stateful MLP code

In this example and according to our identification of processing layers, we differentiate 6 processing layers: packet, flow, user, flows_user, application, and global. There are sections of code of a particular processing layer spreaded over multiple locations (*e.g.* flow layer). Moreover, we can observe nested processing layers (*e.g.* flow, user, and global layers in the first line of "Update_User_Stats"). In Figure 6.5 we present the MLP implementation of the previous code. We insert a migration instruction (with the ID of migration) for every transition to a new processing layer. In Figure 6.6 we present the execution timeline of the "Processing" function. The labels on the top of the picture denotes the processing layers. The dashed lines and number labels indicate the migrations.

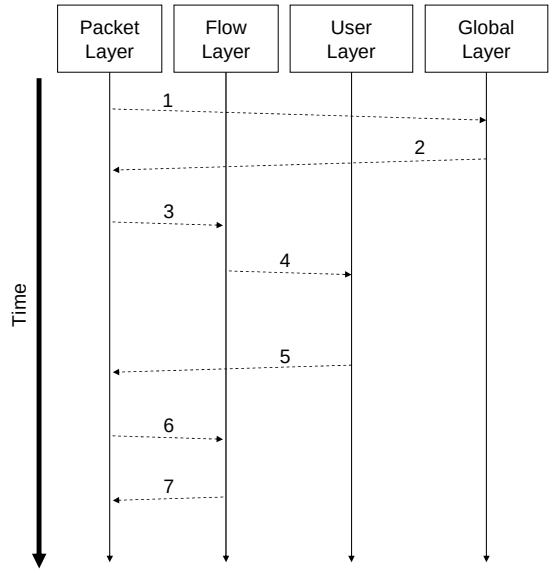


Figure 6.6: Migrations in the ”Processing” Function

6.2.4 Load Balancing

Load balancing is the distribution of the workload and hardware resources in order to maximize the throughput of the system. This issue is sensitive to several factors related to: dependencies among available resources and network processing requirements.

Streams can present dependencies among them due to shared software resources (*e.g.* shared data structure) and shared hardware resources (*e.g.* shared cache). Depending on the sharing degree among streams, the load–balancing mechanism must use different policies to provide quality of service. Otherwise, the system can experience performance degradation.

As an example of this will be when multiple streams are assigned to a particular layer in which most of the time only a single stream can process useful instructions due to conflicts in critical sections. The other streams affect the stream that executes useful instructions if they share hardware resources. In this case the system presents performance slowdown compared to a system that presents reduced number of streams for that particular layer. Some of those negative effects will be discussed in Section 6.4.

Regarding the network processing requirements, the performance requirements can

vary when network traffic behavior changes. There are many factors that may generate important variations in the network behavior (*e.g.* number of active users, applications that handle connetions, time of the day). As we assume that the OS manages the distribution of cores across processing layers, the OS is responsible of collecting and monitoring statistics about processing requirements. Nevertheless, the study of this load balancing issue is out of scope of this thesis.

In addition, one of the weakness of MLP is the lack of a mapping algorithm to distribute workload among multiple cores assigned to a particular layer. Our approach assumes an even distribution of cores to processing layers and threads to cores. We don't use any mechanism to distribute threads according to a value of the packet (*e.g.* all packets of a given flow ID are mapped in the same core). Other authors distribute the workload according to values of the packet contents (*e.g.* flow ID). For example, the threads that process packets of a given flow are mapped to a particular core. This approach leads to unbalanced workload distributions.

6.2.5 Processor Architecture

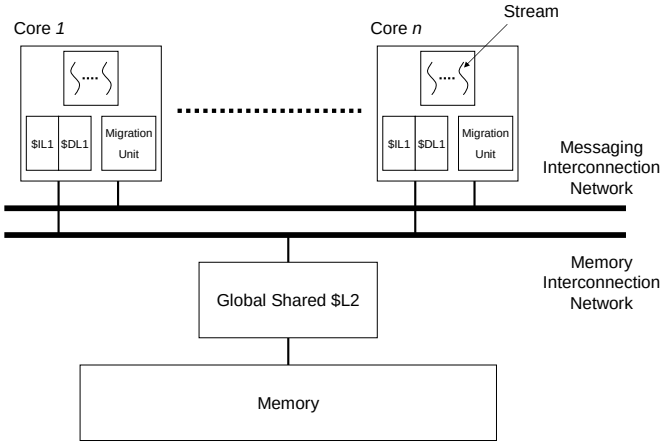
6.2.5.1 Baseline Architecture

Our baseline architecture is based on a generic massive multithreaded architecture as depicted in Figure 6.7(a). The processor comprises a number of processing nodes (Cores) communicated through an interconnection network. Each core includes a processing engine, a local cache system, and a migration unit. A given processing engine contains from 1 to several hardware contexts, called streams. Each core is single–issue in–order five stage pipelined: fetch, decode, execution, memory, write back.

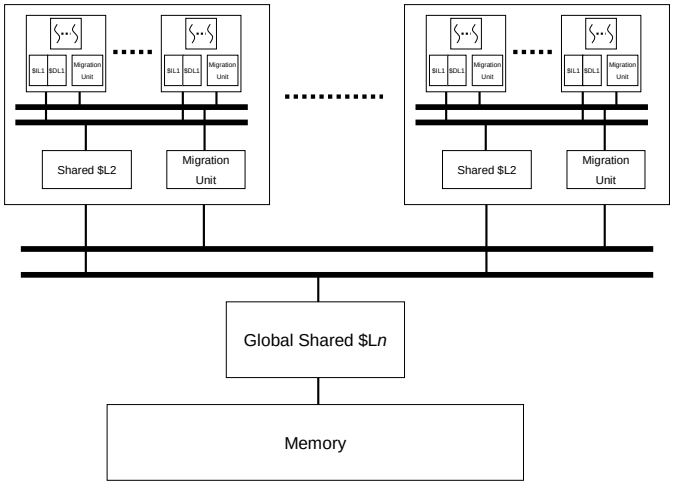
The memory hierarchy design depends on the purpose of the processor. Thus, in order to map our multiple processing layers, it is necessary to have a memory hierarchy as shown in Figure 6.7(b). The hierarchical design increases the benefits of MLP, since processing layers that present dependencies can be assigned to closer levels within the architectural hierarchy.

6.2.5.2 Summary of Commercial Processors for Packet Processing

Some current commercial processors present similar moves to exploit massive multithreaded architectures with different characteristics:



(a) Baseline Processor Block Diagram



(b) Example of an Ideal Processor for MLP

Figure 6.7: Baseline Processor Block Diagram

- **Consentry Networks [18]:** comprises up to 128 cores in a single processor. A similar predecesor processor (Kayamba [37]) contains 256 streams distributed

across 8 cores. Another massive multithreaded architecture also focused on network security is the paper–processor published by Melvin et al. [53]. It includes 8 cores with 32 streams each (a total of 256 streams) simultaneously executed. Each core also integrates an instruction cache, but there is no data cache in the system. Streams access directly to a local or to an external memory. The interconnection network is based on a crossbar that connects all cores. The paper–processor provides support for thread migration and for order–of–seniority packet processing.

- **Cavium Networks [12]:** the Octeon processor integrates up to 16 cores. Each core is dual–issue superscalar with instruction and data caches. Cores are communicated through a synchronization bus and another bus to access to a global shared L2 cache. There are several specialized engines for packet processing (*e.g.* regular expression engines).

- **Sun Microsystems [86, 26]:** the UltraSPARC T1&T2 processors contain up to 8 cores with 4 and 8 streams (a.k.a. strands) per core, respectively. In fact, T2 integrates an additional level of shared resources within each core by separating the streams into two groups of 4 strands each. Strands of a given core are executed in a vertical multithreaded approach (*i.e.* similar than fine–grain multithreading). The cores comprise reduced instruction and data caches. A crossbar connects the cores to a global shared L2 cache. There are no migration on demand capabilities. That is, there is no special migrate instruction, although the streams are moved by the Operating System for load balancing and scheduling purposes. Nevertheless, applications can be distributed across cores communicated by software mechanisms in memory (*e.g.* software buffer).

- **Raza Microelectronics [67]:** the XLR processor presents a multicore multithreaded architecture that integrates up to 32 streams distributed in 8 cores. Each includes reduced instruction and data caches. A ring network communicates cores and caches. Another interconnection network (fast messaging network) provides communication between cores and I/O elements. In addition, the processor also contains a number of accelerator engines for packet processing.

- **Tilera [8]:** the TILE64 processor comprises 64 3–way VLIW cores (a.k.a. tiles) in a single chip interconnected through a multichannel 2D–mesh. The tiles are arranged as a grid of 8x8 tiles. Each tile contains two levels of cache. The aggregation of each L2 cache modul provides a virtual shared L3 cache. The processor provides capabilities to distribute data processing across multiple tiles by using a register–to–register communication protocol.

6.3 Methodology

6.3.1 Traffic Traces

In order to preserve representative features of the traffic, we select a set of public bidirectional traffic traces from the MRA Lab of 622Mbps in the NLANR site [59]. We combine them through the traffic aggregation mechanism proposed in Section 3.4.1. The resulting traffic trace presents nearly 1Gbps with 20K active flows on average. Since the stateful applications deal only with the flow state, the provided aggregation level is enough to represent the low locality of flow states of larger network links.

6.3.2 Workload

We base our studies on different configurations of the Snort–MT (see Section 5.2 for further details) as a representative range of stateful DPI applications. We select and then parallelize a number of stateful preprocessors that show different behavior:

- *perfmonitor:* it collects a wide range of statistics from packet processing intended for network administrators.
- *stream4:* provides TCP stream reassembly and stateful analysis capabilities to track simultaneous TCP streams and to ignore stateless attacks.
- *frag3:* is an IP defragmentation module that applies target-based host modeling anti–evasion techniques for attacks based on information about how an individual target IP stack operates.

Each workload presents a different configuration of enabled preprocessors. Table 6.1 indicates the enabled preprocessors according to the workload identificator. Moreover, in all configurations we employ the default configuration of rule set for rule–matching, that includes a total of 3291 rules.

The right column denotes the identification number of the processing layers available for each configuration of Snort–MT. We identify a total of six processing layers with the following IDs:

1. Packet: the packet contents and internal data related to the packet itself.

Enabled Preprocessors	Workload ID	Processing Layer IDs
Perfmonitor	Mix-1	1, 2, 5, 6
Stream4	Mix-2	1, 2, 3, 6
Frag3	Mix-3	1, 2, 4, 6
Perfmonitor - Stream4	Mix-12	1, 2, 3, 5, 6
Perfmonitor - Frag3	Mix-13	1, 2, 4, 5, 6
Stream4 - Frag3	Mix-23	1, 2, 3, 4, 6
Perfmonitor - Stream4 - Frag3	Mix-123	1, 2, 3, 4, 5, 6

Table 6.1: Workload Mixes

2. Flows: shared information among active network flows.
3. TCP Flows: state of the active TCP flows.
4. Fragmented Packets: state and collected contents of fragmented packets.
5. Stats: global shared statistics about the performance and behavior of the network.
6. Rules: the rule–set database.

We apply a conservative implementation for both MLP and SPL implementation. Thus, the studied MLP code splits the code into six processing layers, whereas the SPL design presents three pipeline stages: decoding, preprocessing, and rule–matching. Further studies on finer–grain implementations are proposed as future work (see Section 7.2.3).

Before collecting statistics, there is a warming stage that presents different number of packets according to the processor architecture configuration. The more active threads distributed across several cores the more packets are processed in the warming stage. On average, we use 5K packets for warming stage per core. We use the same procedure to take measurements in the simulation stage. We employ 30K packets on average per core.

6.3.3 Simulator

We use the SESC simulator [69], a cycle-accurate, execution driven simulator that provides support to simulate multicore multithreaded architectures while running multithreaded workloads. We modify the simulator to support different thread mapping approaches. Migration support have been also implemented in the simulator.

Table 6.2 shows the processor configuration. We base our architecture in the properties discussed in Section 6.2.5.1. We have a number of processing nodes linked to an interconnection network. Each node contains a single issue in–order, 5 stage pipelined core, private instruction and data cache, and the migration logic.

Each core has a configurable number of streams. Register set file is replicated per stream. Pipelined functional units are shared among streams. We employ an static branch predictor that achieves nearly 98% of hit rate according to our studies. Branch miss prediction has 3 cycles of penalty.

Processor Parameters	
# Cores	1-48
# Streams per core	1-8
Fetch Policy	I-COUNT
Issue-width	single-issue
Instruction Buffer	8-entries per stream
IL1 cache	8KB, 4-way, LRU, 32B line
DL1 cache	8KB, 4-way, LRU, 16B line
L2 cache	3MB, 16-way, 12banks, LRU, 64B line
Store Buffer	8-entries per stream
IL1 cache	1 cycle
DL1 cache	1 cycle
Global L2 cache	50 cycles
Main memory	250 cycles
Coherency model	Directory based
Consistency model	RMO
Migration	65 cycles

Table 6.2: Simulation Parameters

Each core has private L1 instruction and write–through data caches. Caches are kept coherent by traking directories in the L2 cache. The write–back non–blocking L2 cache is shared among all cores. The total L2 access latency integrates the latency of crossing the memory interconnection network (*i.e.* cores to shared cache) and the cache access. The simulator follows the specification of Relaxed Memory Order (RMO) consistency model. We assume a latency of 25 cycles as a representative latency of interconnection network for large scale architectures (*e.g.* Tilera64 [8]).

In order to simulate the MLP model we modified the simulator by integrating migration support. We introduce a thread migration instruction with similar support to the paper processor presented by Melvin et al. [53]. The operand of the instruction specifies the destination processing layer. During migration, a single register access per cycle is processed by the stream and sent to the messaging interconnection network. As network processing doesn't require floating point processing, we assume that a migration only transfers the integer architectural registers, unlike other analysis that transfer both integer and floating point register set files [19]. Thus, the migration transfer presents a latency of 65 cycles: 25 for the initiation of the transfer and a cycle thereafter to transmit the architectural registers and additional control information. If there are no idle streams in the destination core, the migrated thread waits in a memory queue until it can be assigned to a stream.

6.4 Evaluation

We first discuss the positive and negative aliasing among threads in both instruction and data caches. Then, we analyze the impact on lock contention through the analysis of conflict rates in critical sections according to the execution model. Finally, we study the performance scalability.

6.4.1 Affinity among Streams

The performance of multithreaded architectures is sensitive to the affinity among streams. Thus, when a thread is assigned to a given stream can produce positive aliasing (sharing instructions and data in the local cache of a given core) or negative aliasing (trashing instructions and/or data in the local cache of a given core).

6.4.1.1 Instruction Cache

Figure 6.8 depicts instruction cache miss rates (Y–axis) in percentages of misses of the total cache accesses per core. Figures 6.8(a), (c), and (e) show miss rates as the number of enabled cores (X–axis) increases in the system. The experiments graphed in those figures assume 4 streams running in each core. Large configurations (more than 6 cores) show flat rates in all models. The RTC shows flat higher miss rate regardless the system

scale, due to the computational complexity of stateful processing. In contrast, SPL and MLP shows higher miss rates in reduced configurations, since there are different pipeline stages or layers mapped in a particular core. SPL and MLP also needs different amount of cores to show flatten miss rates due to they present different requirements to map one–to–one each part of the packet processing (SPL has three pipeline stages and MLP from four to six processing layers).

Figures 6.8(b), (d), and (f) graph miss rates for stream scaling configurations (X–axis). We assume a system with 12 cores. Unlike core scaling, we slightly differentiate slopes among execution models. RTC shows higher increment since streams introduce negative aliasing due to different execution footprints of streams. SPL reduces such negative aliasing because threads running in the same core belongs to the same pipeline stage. MLP shows marginal difference with thread scaling experiments since the negative aliasing is reduced even more. Overall, MLP works better than the other execution models, since there are less variations in code footprint among streams within a given core.

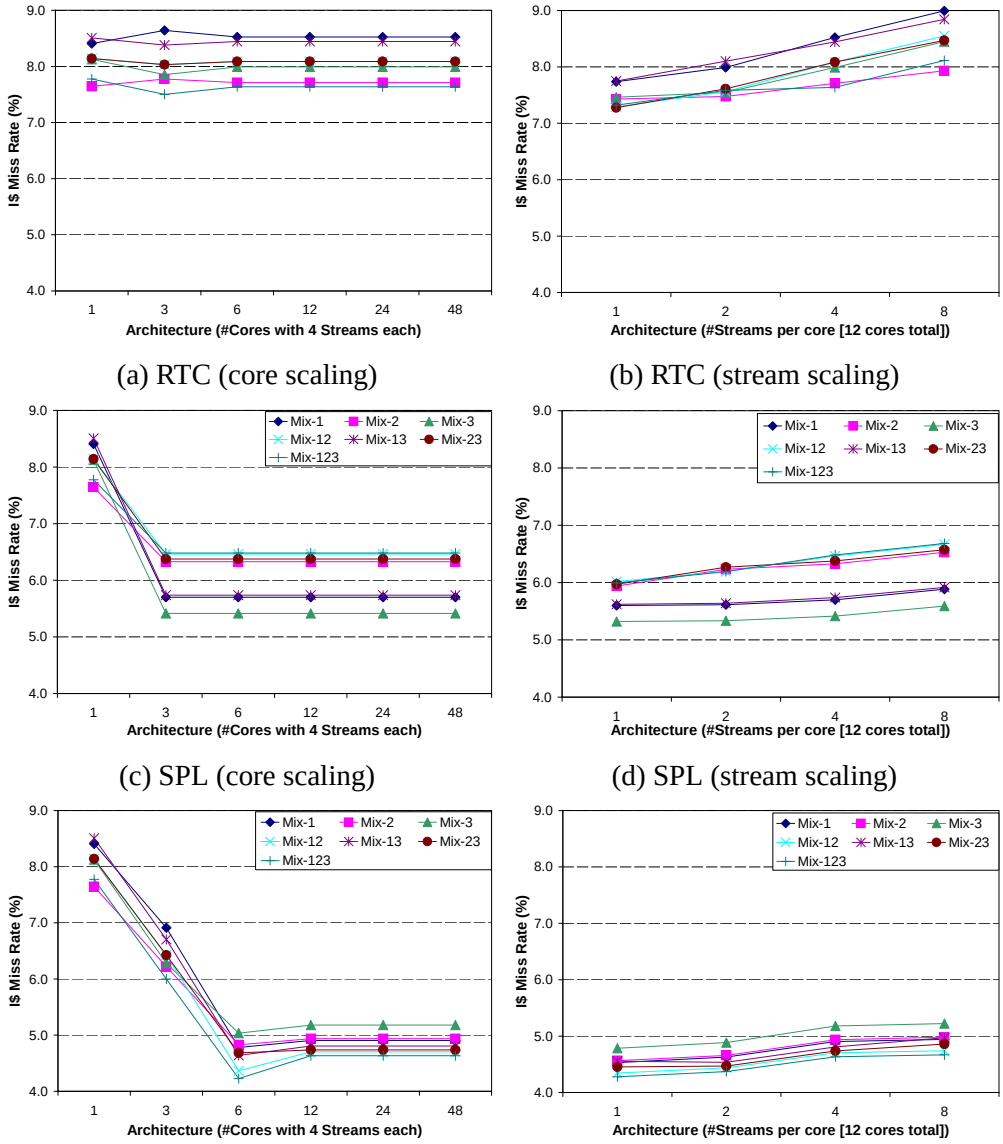
We can observe in Figure 6.9(a) the average miss rates of all execution models. The X–axis denotes the number of streams per core and the number of cores in the system. RTC ranges from 7.4% to 8.4% of miss rate. SPL ranges from 5.7% to 6.3% of miss rate with 3 or more cores. Finally, MLP ranges from 4.5% to 4.9% of miss rate with more than 6 cores. Figure 6.9(b) shows the reduction of miss rates normalized to the I$ miss rate shown by the RTC model. SPL presents about 25% less miss rate than RTC, while MLP show 40% less than RTC.

We would like to remark that MLP shows the best miss rates in an enviroment that can be better tunned for multilevel processing. That is, in applications with optimized implementations for MLP, the instruction cache miss rate can be significantly reduced.

6.4.1.2 Data Cache

In this section we analyze the data cache miss rate. Unlike the studies presented in Chapter 4, we focus the study to the private L1 data cache instead of L2$. The reason behind this is that the main difference among the execution models can be found in the private L1$. The high traffic aggregation keeps low temporal locality of stateful data and our experiments point out that the L2$ show no differences in the stateful data management across the evaluated execution models.

In Figure 6.10 we can observe data cache miss rates (Y–axis) in percentages of



(a) RTC (core scaling)

(b) RTC (stream scaling)

(c) SPL (core scaling)

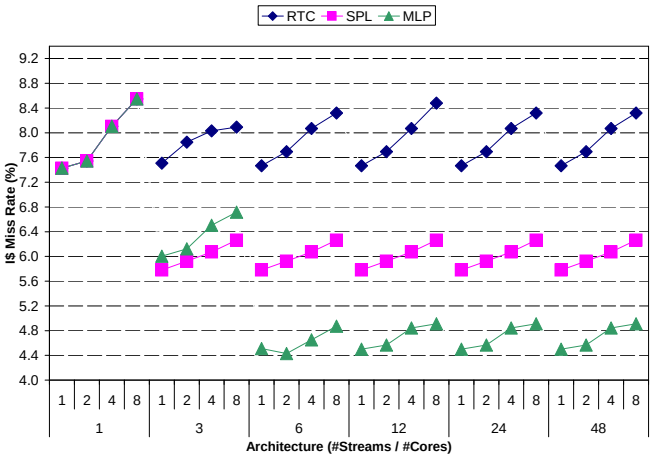
(d) SPL (stream scaling)

(e) MLP (core scaling)

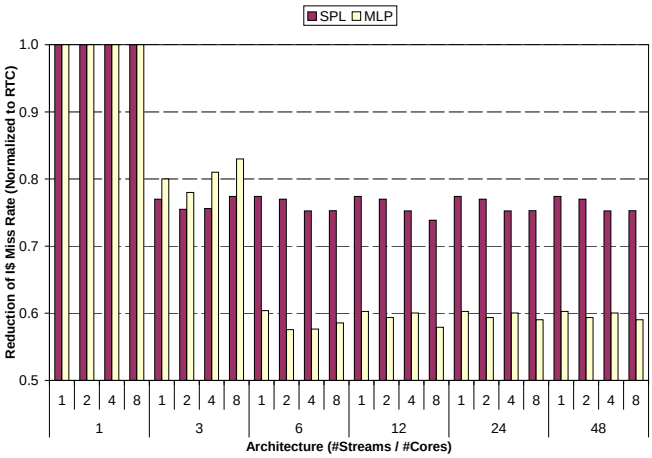
(f) MLP (stream scaling)

Figure 6.8: I$ Miss Rate with Core and Stream Scaling

misses of the total data cache accesses per core. Figures 6.10(a), (c), and (e) show



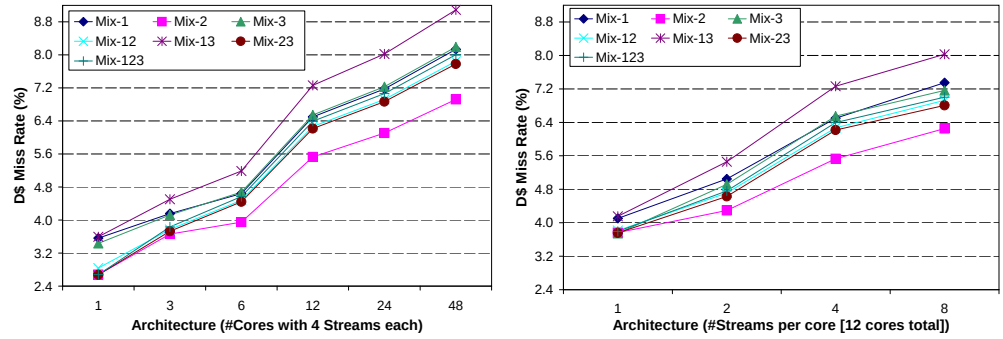
(a) Avg. I$ Miss Rate



(b) Normalized Reduction of avg. I$ Miss Rate

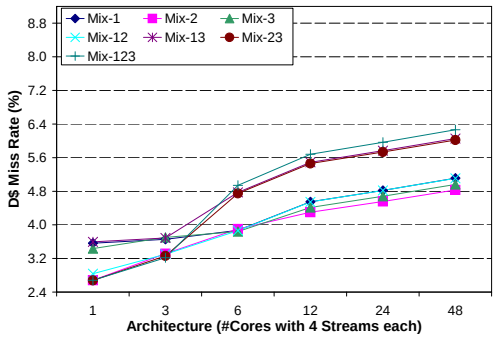
Figure 6.9: I$ Miss Rate Comparison

miss rates as the number of enabled cores (X–axis) increases in the system. The experiments graphed in those figures assume 4 streams running in each core. All execution models show higher miss rates with larger configurations, although RTC seems to show more than linear increments. SPL and MLP are very similar, especially for large configurations, since the same data structures are used in multiple cores assigned to a given pipeline stage or processing layer. As MLP does not apply any data mapping algorithm to select cores within the same level, multiple cores are candidate to be selected and thus

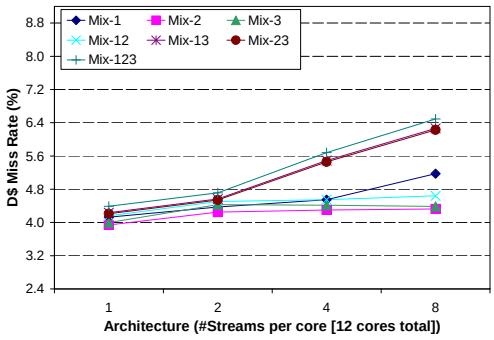


(a) RTC (core scaling)

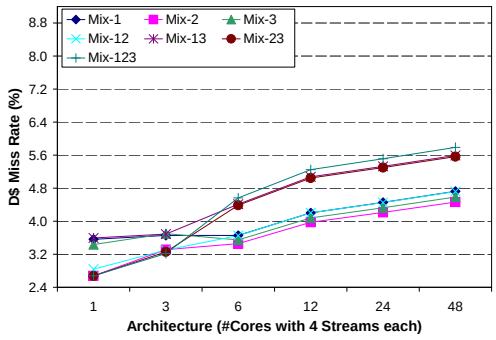
(b) RTC (stream scaling)



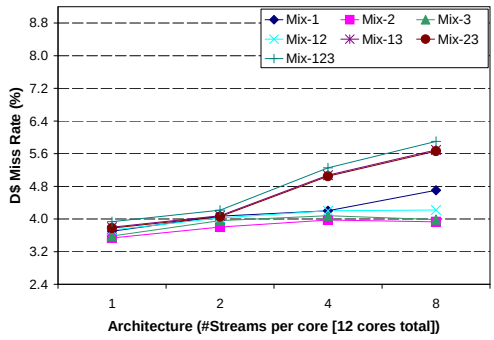
(c) SPL (core scaling)



(d) SPL (stream scaling)



(e) MLP (core scaling)

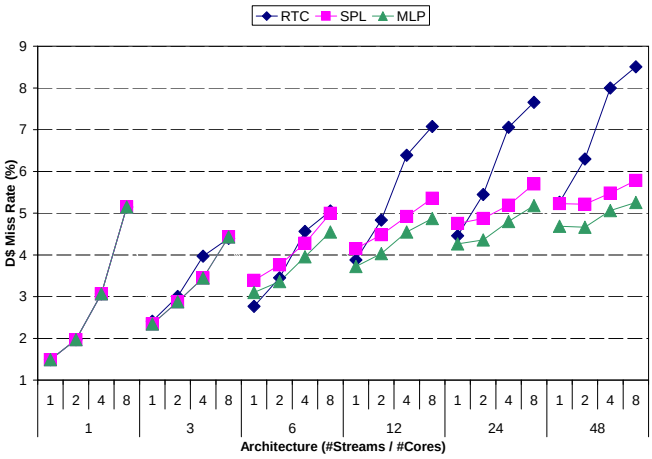


(f) MLP (stream scaling)

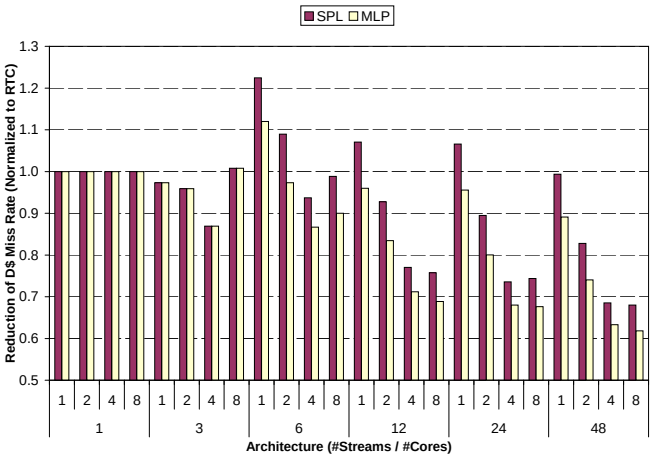
Figure 6.10: D$ Miss Rate with Core and Stream Scaling

negative aliasing cannot be avoided.

On the other hand, Figures 6.10(b), (d), and (f) present miss rates for stream scaling



(a) Avg. D$ Miss Rate



(b) Normalized Reduction of avg. D$ Miss Rate

Figure 6.11: D$ Miss Rate Comparison

configurations (X–axis). We assume a system with 12 cores. RTC shows slightly sharpen slope than the others. That is, multiple threads introduce negative aliasing in cache. MLP shows slightly more flatten results than SPL, since MLP takes advantage of clustering data of a given processing level. However, the positive aliasing improvement of MLP against SPL is not as much as in instruction cache. As we do not apply any distribution policy to allocate data among cores of a given level, MLP shows similar results than SPL. In fact, such policies would have high cost, since data allocation depends on values

of the packet payload.

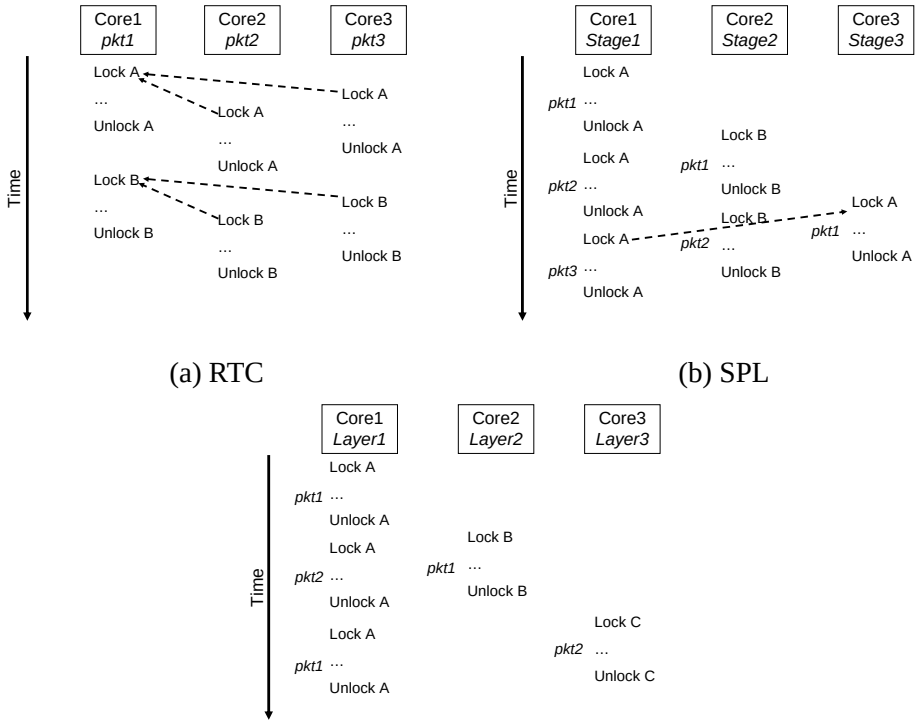
Figure 6.11(a) shows the average miss rates of all execution models. The X–axis denotes the number of streams per core and the number of cores in the system. All execution models show similar results with reduced system configurations of 1, 3, and 6 cores. The differences among models arise with larger configurations. A system with 48 cores shows DL1$ miss rates using RTC model that ranges from about 5.2% to nearly 8.5%. Instead, SPL ranges from 5.2% to 5.8% and MLP ranges from 4.6% to 5.2%. Figure 6.11(b) clarifies the differences among models by showing the reduction of miss rates normalized to D$ miss rate shown by the RTC model. SPL shows higher miss rate than RTC with single streamed cores due to data sharing among cores assigned to a particular pipeline stage. Instead MLP reduces this negative effect. Moreover, MLP reduces nearly 40% the miss rate of RTC compared to 33% of SPL.

Overall, the improvement of MLP is slightly lower than the I$ miss rates due to the sharing of data among cores assigned to the same processing layer.

6.4.2 Lock Contention

During the implementation of Snort–MT we analyze that locks are statically assigned to a particular processing layer. That is, a set of locks usually protects the critical sections of a particular state category. For example, the lock used to protect the hash table of flow states is not used to also protect the global counter of processed packets. Figure 6.12 graphs an example of lock contention in the RTC, SPL, and MLP model. The Y–axis denotes the timeline, while ”Lock x” and ”Unlock x” indicates the lock used in each critical section. The top of each graph, indicates the core IDs and the workload assigned to each core. The RTC model allows that any stream of any core can acquire a particular lock and cause a conflict. The SPL execution model reduce the number of streams that can acquire a particular lock only if the lock is acquired in a single pipeline stage. However, a given lock can be used in multiple pipeline stages. Finally, the MLP model significantly reduces the streams that contend for a given lock, since the model provides a queue–like approach to acquire the lock. The reason behind this is that the locks are related to processing layers. However, this benefit is reduced with the increment of cores assigned to a given processing layer.

Figures 6.13(a), (b), and (c) presents the probability in percentage (Y–axis) that two threads cause a lock conflict (*i.e.* a thread requests a locked lock to access a given critical section) for RTC, SPL, and MLP architectures, respectively. All workloads present



(a) RTC

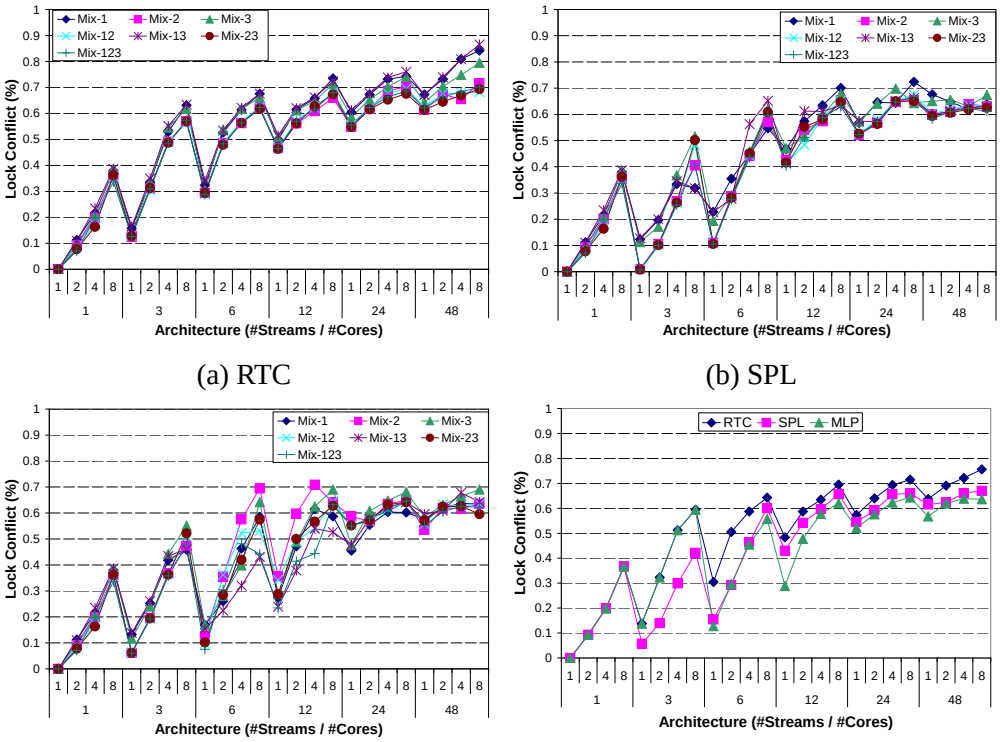
(b) SPL

(c) MLP

Figure 6.12: Example of Lock Contention

similar percentages regardless the architecture configuration. For reduced number of cores we can observe significant impact when increasing the number of streams per core. However, the incremental impact of stream scaling per core is reduced with larger configurations. The average results shown in Figure 6.13(d) manifest that the largest differences between execution models are with architecture configurations of 3, 6, and 12 cores. Larger configurations SPL and MLP presents lower slope than the RTC model.

We can observe in Figure 6.14 the reduction of lock conflict normalized to RTC rates (Y–axis) using SPL and MLP. A system with 3 cores shows large conflict reduction using SPL than MLP, since MLP doesn't properly map the cores to processing layers because there are more processing layers than cores. Thus, the benefits of core specialization are reduced. However, with larger architecture configurations MLP works better than SPL, because the model reduced the number of threads contending for a given lock. The most significant difference, especially with RTC model, is shown by the configuration with 6



(a) RTC (b) SPL

(c) MLP (d) Averaged Rates

Figure 6.13: Lock Conflict Rates

cores and 1 stream per core, because cores are mapped one–to–one to processing layers. With this configuration MLP presents about 0.67 lower lock conflict than the RTC model. With larger configurations MLP reduces about 10% the lock conflict probability of RTC model and about 5% of the SPL model.

6.4.3 Performance Scalability

To better understand the analysis of performance scalability, in Figure 6.15 we compare the occupancy time per core (Y–axis). We define occupancy time as the time that a given core is executing at least one stream, although it is waiting to acquire a lock of a particular critical section. The core is not active when there are no threads assigned to

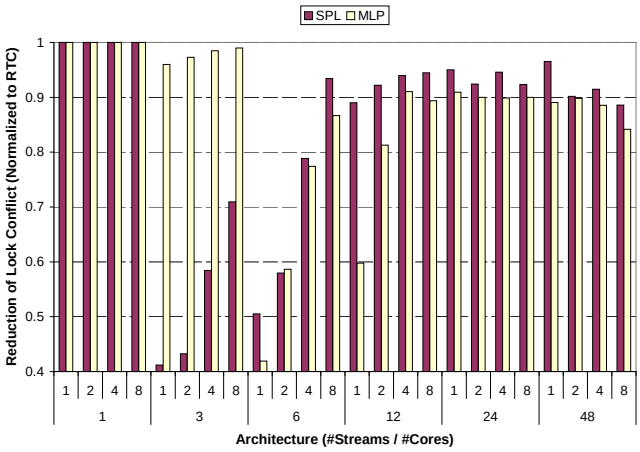


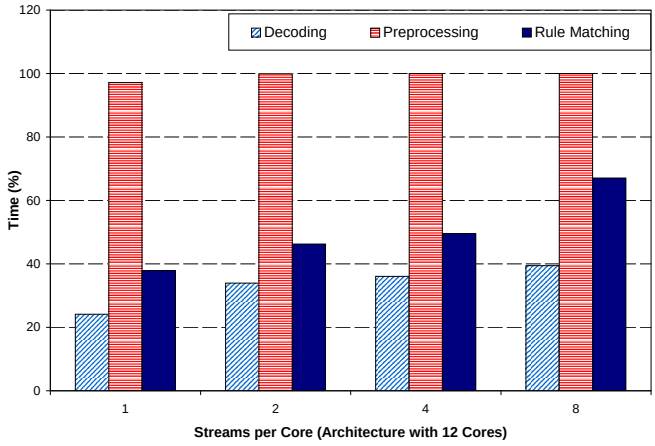
Figure 6.14: Normalized Reduction of Lock Conflict

the core (*i.e.* all threads migrate to remote cores). We show average metrics for Mix–123 workload, because it maximizes the exploitation of all processing levels and pipeline stages. The X–axis indicates the system configuration with the number of streams per core, assuming a system with 12 cores. RTC is not shown in the graph since all cores are active 100% of the time.

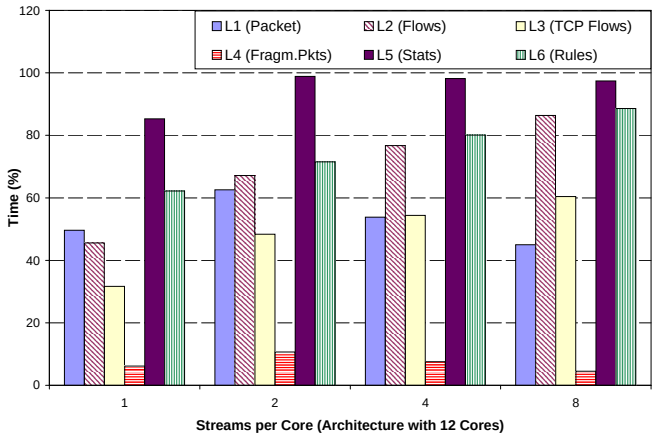
In the top graph, we can observe that SPL presents nearly 100% of core utilization, due to preprocessing stage (bars with horizontal lines). This stage comprises most of the packet processing time. Rule matching stage (the solid bars) increases occupancy rate from 37% with single stream cores to 67% with 8 streams per core. In contrast, the cores assigned to decoding stage show the minimum core utilization of all pipelined processing ranging from 24% to 39%. As a consequence of high processing contention localized in preprocessing stage, the remaining pipeline stages show reduced occupancy rates.

In Figure 6.15(b) we can observe that MLP presents wide variety of active time distribution. Layer ”L4 (Fragmented Packets)” shows the lowest core utilization with over 10% in all configurations. Instead, the layer ”L5 (Stats)” presents the highest occupancy rate in all cases with about 98% with multithreaded cores. In fact, all processing layers increase core utilization while the number of streams per core increases, except for the processing layer ”L1 (Packet)” that keeps about 45% of active time.

Nevertheless, stream occupancy has to be analyzed in order to break down the core



(a) SPL model



(b) MLP model

Figure 6.15: Average Occupancy Time per Core

utilization. Table 6.3 presents stream occupancy time rates normalized to the core utilization previously discussed. The most left column denotes the core configuration in terms of streams per core. For each configuration we show the percentage of the time a number of streams (indicated by the second column) are active. The total sum of all rates for a given configuration and a particular column must be 1. For example, the dual stream cores assigned to ”L1 Layer” of MLP execution model use 40% of the time only one thread. The two threads are active the remaining 60% of the time.

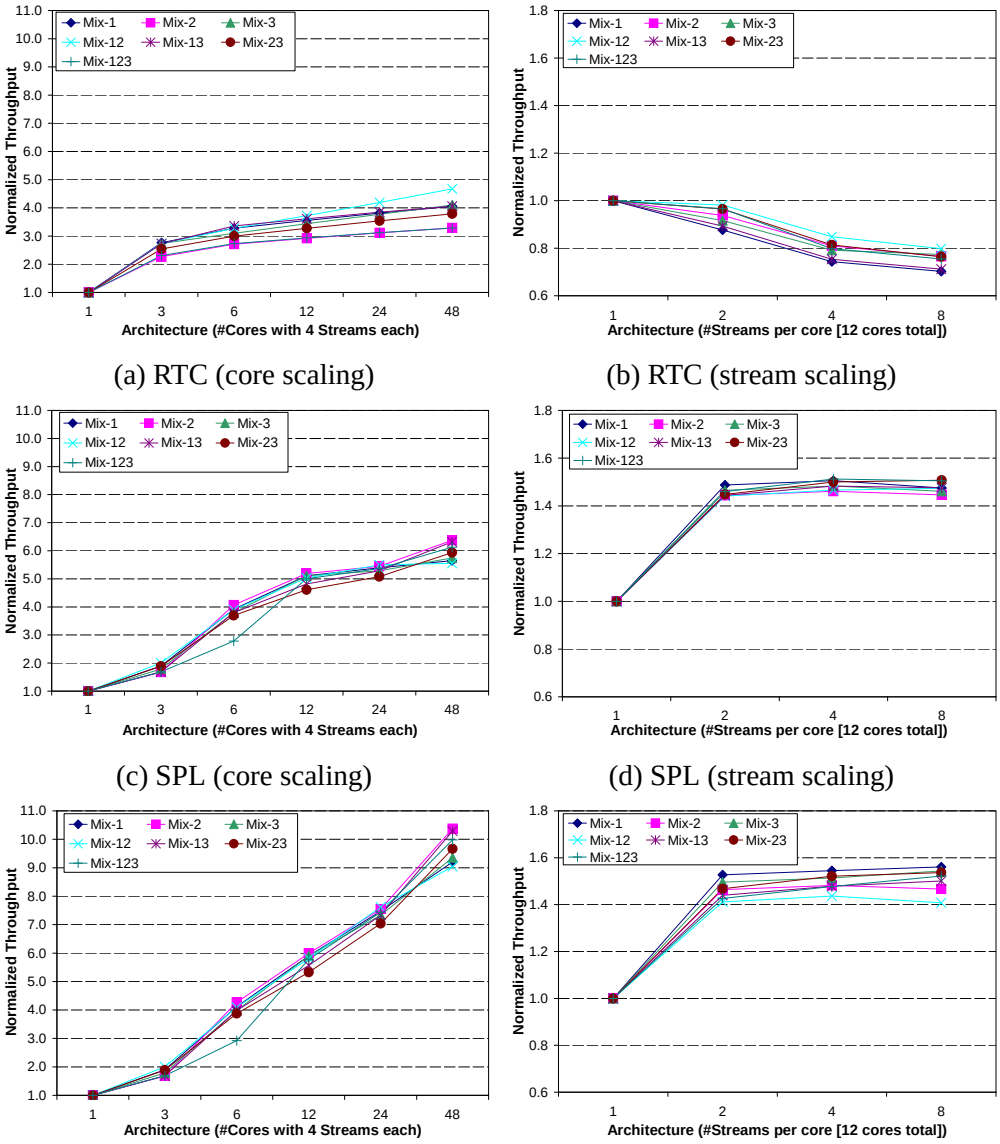
Stream Occupancy Rate											
		MLP						SPL			RTC
Config.	#Strs.	L-1	L-2	L-3	L-4	L-5	L-6	Decod.	Preproc.	Rule Mat.	All
1Str.	1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2Strs.	1	0.40	0.40	0.73	1.00	0.03	0.39	0.86	0.00	0.67	0.00
	2	0.60	0.60	0.27	0.00	0.97	0.61	0.14	1.00	0.33	1.00
4Strs.	1	0.37	0.47	0.91	1.00	0.22	0.70	0.65	0.00	0.65	0.00
	2	0.36	0.38	0.07	0.00	0.39	0.26	0.25	0.00	0.25	0.00
	3	0.18	0.14	0.01	0.00	0.31	0.04	0.08	0.00	0.07	0.00
	4	0.09	0.01	0.01	0.00	0.08	0.00	0.01	1.00	0.03	1.00
8Strs.	1	0.28	0.17	0.63	1.00	0.07	0.37	0.54	0.00	0.55	0.00
	2	0.17	0.20	0.20	0.00	0.10	0.29	0.24	0.00	0.27	0.00
	3	0.15	0.19	0.09	0.00	0.14	0.18	0.12	0.00	0.13	0.00
	4	0.12	0.17	0.03	0.00	0.18	0.10	0.05	0.00	0.04	0.00
	5	0.08	0.12	0.02	0.00	0.18	0.06	0.02	0.00	0.01	0.00
	6	0.08	0.08	0.01	0.00	0.14	0.00	0.01	0.02	0.00	0.00
	7	0.05	0.04	0.01	0.00	0.09	0.00	0.00	0.03	0.00	0.00
	8	0.06	0.03	0.02	0.00	0.10	0.00	0.00	0.94	0.00	1.00

Table 6.3: Distribution of stream occupancy per core

As RTC doesn't migrate threads, all streams are always active regardless the system configuration. For dual stream cores, we can see that SPL shows from 67% to 86% of the time running a single thread in both decoding and rule matching stages. Even scaling the number of streams per core in both stages, 90% of the time only are active two and three streams. Instead, preprocessing stage always requires full utilization of streams regardless the configuration. Thus, the system doesn't take advantage of large stream configurations and throughput will not be significantly increased for those cores.

The MLP model shows about 60% of the time layers "L1", "L2", and "L6" exploit utilization of two threads in dual threaded cores. These layers increase stream utilization in 4–stream cores and 8–stream cores. Nevertheless, in the largest configuration they only use two streams over 50% of the time on average. In addition, "L3" requires fewer streams, since it uses one stream nearly 75% of the time, regardless the core configuration. With largest configurations it shows similar occupancy than decoding and rule matching SPL stages, due to the coarse–grain implementation of MLP. "L4" presents very lightweight requirements, because fragmentation packet shows reduced processing workload due to the properties of our network traffic. Finally, "L5" is the processing layer that exploits occupancy of larger configurations.

Figure 6.16 depicts the performance scalability for RTC, SPL, and MLP processing



(a) RTC (core scaling)

(b) RTC (stream scaling)

(c) SPL (core scaling)

(d) SPL (stream scaling)

(e) MLP (core scaling)

(f) MLP (stream scaling)

Figure 6.16: Performance scalability

approaches. The results are normalized to the smallest configuration in both core and stream scaling graphs. Figure 6.16(a) graphs RTC performance scalability with core

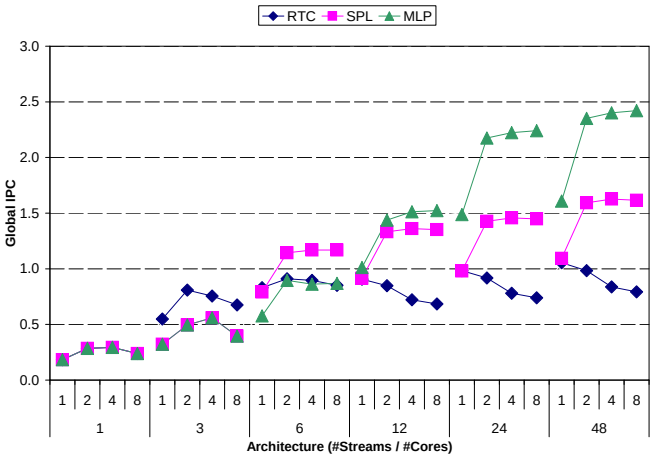
scaling. We can observe that systems with 3 cores provide nearly 2x of speedup compared to a single core system. However, scalability is flatten for larger systems showing 4x of speedup for systems with 48 cores.

In contrast, SPL and MLP present slightly lower speedup for systems with 3 cores, but sharpen scalabilities for larger configurations in Figures 6.16(c) and (e), respectively. In fact, SPL shows almost 6x of speedup with 48 cores system, while MLP presents over 10x of speedup.

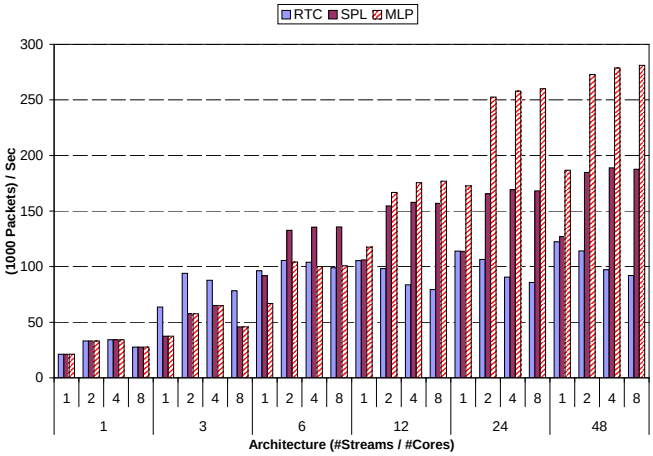
Performance scalability with stream scaling can be seen in Figures 6.16(b),(d), and (f) for RTC, SPL, and MLP, respectively. RTC shows slowdown as we increase the number of streams per core up to 20% with a maximum of 8 streams, due to instruction and data cache trashing. On the contrary, SPL and MLP show similar speedup regardless stream scaling. In fact, we observe speedup of 1.4x with 2 streams for both processing approaches. But the non–scaling performance is due to the occupancy rates for large configurations previously discussed (*i.e.* conflicts on critical sections and distribution of workload).

Figure 6.17(a) and (b) graphs the average system throughput in terms of total IPC and thousands of packets per second (KPPS), respectively. We can observe in Figure 6.17(a) that RTC shows similar behavior and performance than SPL and MLP for reduced architectures. As configuration increases SPL and MLP present higher performance than RTC. In fact, the MLP model shows the highest performance in systems with more than 6 cores. We go from IPC of 0.18 in a single stream single core system to maximum IPCs of 2.42, 1.6, and 1.05 in the best performance system configuration of MLP, SPL, and RTC models, respectively.

Figure 6.17(b) translates total IPC to KPPS metric. While single stream single core architecture is able to provide nearly 21 KPPS, largest system configurations show about 281 KPPS, 188 KPPS, and 122 KPPS for MLP, SPL, and RTC models, respectively. We can see that MLP systems is the only model able to provide more than 200 KPPS that is representative requirements for 1Gbps networks.



(a) Avg. IPC scalability



(b) Avg. Packets per Second scalability

Figure 6.17: Throughput scalability

6.5 Related Work

6.5.0.1 Parallel Network Processing

Wolf et al. [97] analyzed network workloads according to resource utilization. They employed annotated directed acyclic graphs (ADAG) in order to capture runtime properties,

such as number of instructions, memory reads and writes, and dependencies between instructions and clusters of instructions. ADAG should allow to identify parallelizable application parts. Several methodologies have been proposed in order to map workloads on multicore network processors according to ADAGs profiling results [98] or other profiling analysis [24, 101, 48, 65, 83]. However, current proposals are focused on layer 2-3 applications in order to maximize resource utilization. These algorithms, however, are too simplistics for DPI workloads. They do not take into account multiple levels of stateful data sharing nor the complexity of DPI applications.

Hayes and Luo presented a compact finite automata, called DPICO, to provide high speed deep packet inspection [28]. Kim et al. [38] proposed another system architecture for high speed DPI in signature based network intrusion prevention. Both architectures address pattern matching scalable optimizations to achieve high speed processing over 10Gbps. Hardware support for deep packet inspection has also been proposed by Yu et al. [106]. However, the authors deal with rule–based intrusion detection rather than statefulness and other deep–processing tasks.

Paxson et al. [91] proposed a NIDS cluster in order to scale performance while sustaining distributed stateful processing. Colajanni and Marchetti [17] employed several distributed sensors to propose parallel architectures for stateful intrusion detection. In this case, they migrate a given state from one system to another to sustain scalability while states and alerts are correctly maintained. Previously, Kruegel et al. [41] also proposed distributed sensors architecture to provide stateful processing for high speed networks. Additionally, Paxson et al. [64] also proposed multicore architecture to sustain stateful packet processing at high speed networks. The system distributes threads according to state location to enhance cache locality. However, those proposals don't distinguish among multiple stateful data categories.

Regarding Snort few authors have previously proposed parallel designs and analyses. Haagdorens [27, 94] presented several multithreaded designs of Snort. However, they don't assign synchronization mechanisms to shared data structures, but preprocessors. Then, reduced performance scalability is achieved with those approaches and no further advantages are obtained from massively multithreaded architectures. In fact, the authors evaluated performance scalability using reduced parallel architectures and show low throughput. Recently, Schuff et al. [77, 76] have analyzed and proposed parallel designs of Snort running in parallel architectures, but there is no information about the synchronization mechanisms employed to protect shared data structures. Moreover, they distributed flows across cores according to the flow ID to increase temporal locality of data structures and reduce the probability of dependencies due to order–of–seniority,

since all packets of a given flow are processed by a given thread. Our studies show that high aggregation traffic networks show low temporal flow locality and thus, reduced locality improvement will be achieved. Additionally, the authors only measured throughput for reduced parallel architectures, but they did not analyse neither memory performance nor parallelism limitations.

6.5.0.2 Distributed Workload in Parallel Architectures

Yeung et al. proposed a similar processing approach, called Multigrain Architectures [105]. They differentiate fine and coarse grain sharing, in the sense of temporal and spatial data locality, to heterogeneously exploit parallelism in distributed shared memory systems. Workload is distributed in order to exploit fine grain cache line sharing within each parallel workstation and coarse grain page sharing across parallel workstations. Our concept of multilayer is similar to multigrain, but the differences are twofold. First, our concept of processing level parallelism related to a particular layer processing is based on dependencies among packets, instead of dependencies among different parts of a single parallel workload processing. And second, as first design step, the conceptual processing layers are identified by the application developer in terms of critical section (fine grain) and sets of functions (coarse grain) categories.

The RAW architecture [87] is the basis of the recently commercialized TILE64 processor [8] that statically partitions code onto multiple tiles and statically orders communications between these tiles so as to lower communication delay, enabling fine-grained parallelism. This architecture is suitable for stream processing by exploiting low overhead communications in the high performance 2D-mesh network [99]. Other proposed architectures, such as TRIPS [74] and Smart Memories [47] can be dinamically configured to suit and exploit different application classes and granularities. However, they are not addressed to detect and exploit granularities among different packets. We add some functionality to similar multicore architecture, but we address parallelism exploitation analysis for stateful DPI properties, which show different dependency features than general applications. If a particular processing layer is more stressed than others due to variation of network traffic, OS can monitor stats and re–assign cores to processing layers by updating its own lookup tables.

Chen et al. proposed Network-Driven Processor (NDP) as a methodology to exploit parallelism by partitioning workload with support for thread creation, scheduling and context-switching in order to reduce mapping overheads, maintaining energy and speed-balancing among cores [14]. Although they show similarities in the architecture, our

methodology goes one step further. It does not create threads, but it performs migrations between cores to enhance positive aliasing among threads. In addition, they are focused on other types of applications, instead of network based workloads that present features related to network traffic. Thus, we assign cores to processing layers, unlike their hardware support that does not show this concept.

Marty and Hill [50] proposed virtual hierarchies as a way to enhance positive interaction of several virtual machines mapped to different cores. Their main concern is focused on optimization of memory system to maximize shared memory accesses within a given virtual machine while minimizing the interference due to memory accesses from other virtual machines. Our methodology is similar in the sense of assigning cores to processing layers instead of virtual machines. However, we exploit other parallelism properties of the workload, introduced by the programmer and identified with the network traffic knowledge.

6.6 Chapter Summary

In this chapter we have presented a new processing paradigm to maximize throughput of massive multithreaded architectures running stateful parallel applications, called MultiLayer Processing. According to the characteristics described in Chapter 5, there are nested levels of critical sections and significant percentage of code within critical sections. In fact, it is likely that applications of the near future will enhance the features described in this thesis.

The code of parallel stateful applications can be categorized in layers similar to the TCP/IP network layer stack, so that each layer is an exploitable parallelism level. We have presented MLP as an execution model able to exploit the wide variety of parallelism levels of stateful workloads, such as packet, flow, application, user. We have described the user–defined layer detection methodology as well as a generic MLP implementation with software and hardware support. The main goal of this chapter is to validate the performance scalability of MLP as an alternative to improve parallel stateful processing.

Our analyses validate that MLP increases positive aliasing and reduces negative stream interefences in private instruction and data caches in multicore multistream architectures. We show that MLP reduces up to about 40% the instruction cache miss rate and up to 30% the data cache miss rate compared to the RTC model. The penalty of this technique is the cost of migration. However, our experiments show there are no

significant cold cache negative effects in the destination core.

Finally, lock conflicts in a non–optimal MLP implementation reduces about 10% compared to the RTC execution model and about 5% against the SPL model. For this reason, we think that MLP is a good alternative employing optimized parallel code with multiple processing layers like stateful applications. Nevertheless, even with a non–scalable implementation using pthread POSIX, performance scales better with MLP than RTC and SPL models. In fact, MLP is the only processing approach able to provide required throughput for 1 Gbps networks (*i.e.* more than 250 KPPS) even using a system with 48 streams. MLP allows better scalability of parallel network DPI workloads and we think it can be adapted to other network–like parallel workloads.

Chapter 7

Conclusions and Future Work

This thesis provides the first analysis of stateful applications as well as proposes the first processing approach to overcome the bottlenecks of parallel stateful packet processing running on massive multithreaded processors. This chapter summarizes the main contributions of the work in Section 7.1 and outlines some future work in Section 7.2.

7.1 Main Contributions

This thesis comprises three sets of contributions, namely: network traffic analysis, workload characterization, and architectural support to exploit parallelism.

7.1.0.3 Network Traffic Analysis

Chapter 3 presents the first twofold network traffic analysis in the area of stateful packet processing. Firstly, we have demonstrated that stateful processing is not sensitive to the loss of IP address distribution due to traffic sanitization, unlike other applications of layer 2–3. Without this conclusion, experimental simulations could not be done because publicly available traces have their IP addresses sanitized.

Secondly, we have presented the impact of traffic aggregation on the memory performance of network applications. To do this study, we have also developed a methodology

to aggregate network bandwidth starting from narrower bandwidth traces, while keeping representative network properties. We assume close to linear increment of traffic aggregation. This behavior actually depends on several network parameters. Nevertheless, the close–to–linear assumption can be representative of real environments. The analysis concludes that stateful applications are more sensitive to traffic aggregation than other network workloads. Especially in the L2 cache, since it is not capable of keeping stateful data, due to the reduced temporal locality of flow states within a window of packets. Other layer 2–3 applications can be also sensitive with large shared data structures, such as the IP forwarding table. In contrast, the impact is lower compared to the stateful workloads.

7.1.0.4 Characterization of Network Applications

We have applied the results of network traffic analysis to properly characterize network workloads in Chapter 4. We have proposed a new classification based on the data management of packet processing:

- Self–Contained: each packet contains all data needed for processing.
- Stateless: shared data structures are used to check packet data and trigger an action.
- Stateful: the application keeps track of packet processing in order to provide higher knowledge about the packet processing.

Workload characterization is performed on single threaded applications to better analyze the application behavior and reduce the interaction effects of multithreaded execution. We have compared workload categories focusing the analysis on application performance as well as ILP, cache performance, and branch prediction. The main results show that stateful applications present significantly lower performance than self–contained and stateless applications. The main reasons behind this are the reduced ILP and large percentage of long latency memory accesses due to the statefulness (e.g. flow state, application state). Stateful processing requires to keep large amount of memory for state data structures that present low temporal locality due to the current and near future network traffic characteristics. Upper layer states, such as user state, can present higher temporal locality, but other issues arise (e.g. higher contention on critical sections, stronger likelyhood of packet dependencies, larger data structure size, probably

lower spatial locality). Instead, stateless and self-contained applications present marginal cache miss rates.

In addition, we have presented the first characterization of stateful processing during the lifetime of network connections. We have shown that the processing and memory requirements are sensitive to the position of a packet within the TCP connection lifetime. There are applications that present irregular workloads. For example, security network services can show the highest requirements in the first packets of a connection. But the requirements are reduced once the application marks the connection as safe or attack hazard. On the contrary, other stateful workloads, such as monitoring or billing, show constant processing requirements among packets. The results of this chapter point out key features for understanding the behavior and requirements of stateful processing in order to properly overcome architectural bottlenecks.

We have devoted a great effort to parallelize the most well known stateful DPI benchmark, called Snort, to evaluate parallel architectures. Although it is a minor contribution of this thesis, the resulting Snort Multithreaded (Snort–MT) application is a key tool for the research community because there are no publicly available multithreaded implementations. Other authors have presented different designs, but this is the first implementation that is addressed to run on heavy parallel architectures.

7.1.0.5 MultiLayer Processing

In Chapter 5 we have introduced key knowledge about parallel stateful processing for better understanding the parallelism limitations as well as the throughput requirements to sustain a given network traffic bandwidth. We emphasize the significant percentage of code within multiple nested levels of critical sections. Although Snort–MT presents up to three levels of nested critical sections, it is likely that more complex stateful DPI applications can present deeper levels of nesting. The main drawbacks of current parallel network processing approaches can be summarized in two issues: high negative interaction among threads and fuzzy identifiable processing pipeline stages.

In Chapter 6 we have proposed a new execution model, called MultiLayer Processing (MLP), to properly exploit the parallelism of stateful workloads while overcoming the above mentioned issues. The analysis of Snort–MT presents multiple layers of data management (*e.g.* packet, flow, application, user). The application can be split in sections of code according to the handled layer. In this dissertation we assume that the application

developer identifies coarse grain processing layers. These layers show processing similarities among packets in both code and data. It is common in stateful applications that a given layer is spreaded over the packet processing timeline.

In the MLP approach, the thread that processes a particular packet migrates from one core to another according to the layer that is processed. The main advantage of the layer based migration is that it is not sensitive to the spreading of a given layer over the packet processing timeline. On the contrary, software pipelining cannot gather such properties in a single pipeline stage. Nevertheless, the more distributed layers the more penalty can arise due to migrations.

We have presented the software and hardware support to implement MLP in a massive multithreaded architecture. OS initially assigns cores to processing layers as well as it is in charge of system control management for required reassignment. Hardware support migrates threads by selecting one of the assigned cores to the destination processing layer.

We have compared MLP against current network parallel execution approaches: run–to–completion (RTC) and software pipelining (SPL). Our results point out that MLP is the approach that better reduces negative aliasing of threads in the instruction cache. Even though negative aliasing in data cache is also reduced in the SPL approach, MLP presents a higher reduction due to its finer–grain task balancing. The main reason behind the SPL lower improvements is the inability of a thread to optimally map to a core within a given processing layer. Another important benefit of MLP is the ability to reduce the contention on a given critical section, since it provides an inherent serialization of critical section execution. MLP is able to reduce up to 20% the probability of lock conflict of RTC. However, it only shows 5% less than the SPL approach due to the coarse grain implementation of the application. We have also analyzed thread and core utilization and its impact on performance scalability. The results show that MLP scales better in parallel architectures than the other approaches. In large configurations (*i.e.* 192 threads distributed in 48 cores of 4 threads each) MLP sustains 20% of throughput scaling efficiency, while RTC and SPL present over 10%. For example, while MLP provides enough throughput with 48 threads to sustain stateful DPI service at 1Gbps (more than 250 KPPS), 384 threads are not enough for SPL and RTC to provide the required throughput.

7.2 Future Work

We have discussed the current trend to exploit network bandwidth and to provide more complex stateful DPI services in the Internet. We are confident that this trend will be steeper in the near future. For this reason, there are several research topics that are an interesting follow-up to this thesis.

Further analysis of network traffic, gradual changing of parallel programming models, and additional proposals to improve MLP approach are some of the research areas we believe that are important to focus on. In fact, we are already working on some of these topics. Furthermore, the resulting knowledge of this thesis has lead to do other research works.

7.2.1 Network Traffic Processing Issues

As pointed out in the introduction of this thesis, new network services are emerging in the Internet. These services employ new methodologies to manage network connections leading to different patterns of traffic workload. We think that further characterization of network traffic is required to understand the evolution of the Internet and its implications.

The evaluation of systems for network processing under different traffic and stateful DPI scenarios arises several difficulties. There is a lack of available traffic traces that present the required range of network features. In addition, there is also a lack of different stateful applications. We think that both limitations can be overcome by developing analytical models that mimic a variety of traffic and stateful DPI behavior. This thesis provides charaterization studies that can be used as the basis of analytical models.

7.2.2 Parallel Programming Tradeoffs

We strongly believe that new kinds of stateful require to employ optimized parallel programming techniques to exploit parallelism while providing the required order–of–processing restrictions. Current parallel programming models (*e.g.* transactional memory) provide most of the parallel requirements. But, there is still room for improvement for order–of–processing requirements, such as parallel execution of transactions according to the packet order–of–arrival. We think that future stateful applications will present more requirements regarding the processing order.

Another important topic to be addressed is the compiler cooperation for enhancing the inherent parallelism of stateful packet processing. It is very complex to detect all levels of parallelism of such applications, due to the complexity of workloads and required network knowledge to properly exploit the processing layers. Many times the programmers under–exploit, undetect, or even incorrectly use parallelism in the application. Thus, it would be very useful to focus on compiler techniques to overcome the weakness of network parallelism exploitation.

Finally, although we develop an scalable parallel design of Snort, we think it is necessary to propose other implementations using different parallel programming models. This topic will provide useful benchmarks for doing research in network processing and parallel programming models. We think that oncoming parallel stateful applications will be a representative pool of benchmarks for parallel programming issues. Even some initial research has been started on this direction.

7.2.3 MultiLayer Processing

This thesis presents a new parallel processing approach, called MultiLayer Processing, and we propose an implementation on a massive multithreaded architecture. However, we believe there are four interesting research topics in this area.

- We have proposed a general implementation of the MLP architecture making the OS responsible for managing lookup tables of the processing nodes. However, it is necessary to have knowledge about the network traffic properties and workload requirements in order to properly reassign cores to processing layers. It would be good to propose a mechanism that gather such information to dynamically modify the lookup tables and core assignment. Moreover, the dynamic rebalancing of the system will lead to performance impact as we discussed in Section 6.2.4.

- The proposed MLP implementation needs some information from the programmer, such as number of processing layers, allocation of migration instructions, and the destination processing layer of a migration. The programmer needs to have a good global vision of the application in order to identify the mentioned processing layers. Therefore, it would be very interesting to develop techniques that let the architecture dynamically detect and exploit MLP without the information from the programmer. That is, a mechanism that automatically detects the processing layers that can be exploited.

- Our experimental environment is based on a conservative MLP parallel version of Snort. We think it is necessary to devote some effort to develop eagger MLP designs to evaluate other tradeoffs. For example, migration dead–lock hazards, improvement of MLP running workloads with wider range of layers, and performance degradation due to high migration frequency.

- This thesis is mainly focused on analyzing the performance scalability of MLP, but it does not provide an analysis about area and power consumption of the MLP designs. We are concerned on such issues as well as other parallel architectural tradeoffs (*e.g.* on chip interconnection bandwidth) that can impact on MLP performance.

Bibliography

[1] OpenMP Organization. *http://www.openmp.org*.

[2] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. April: a processor architecture for multiprocessing. *SIGARCH Comput. Archit. News*, 18(3a):104–114, 1990.

[3] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera Computer System. *Proceedings of the 4th Intl. Conference on Supercomputing*, pages 1–6, June 1990.

[4] AMD. http://www.amd.com, 3DNow! technology manual. In Technical Report, 1999.

[5] Scientific Ring of Catalonia. *www.cesca.es/comunicacions/anella.html*.

[6] Argus - auditing network activity. *http://www.qosient.com/argus*.

[7] Eduard Ayguade, Xavier Martorell, Jesus Labarta, Marc Gonzalez, and Nacho Navarro. Exploiting Multiple Levels of Parallelism in OpenMP: A Case Study. In *ICPP '99: Proceedings of the 1999 International Conference on Parallel Processing*, page 172, Washington, DC, USA, 1999. IEEE Computer Society.

[8] Max Baron. Tilera's cores communicate better. *Microprocessor Report*, Nov 2007.

[9] Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 282–293, New York, NY, USA, 2000. ACM.

[10] J. Beale, J. C. Foster, J. Posluns, and B. Caswell. *Snort 2.0 Intrusion Detection*. Syngress Publishing Inc., 2003.

[11] Cooperative Association for Internet Data Analysis. *http://www.caida.org*.

[12] Cavium Networks Inc. *http://www.caviumnetworks.com/pdfFiles*.

[13] P. Chandra, F. Hady, R. Yavatkar, T. Bock, M. Cabot, and P. Mathew. Benchmarking network processors. In *Proc. of 1st Workshop on Network Processors - NP1, Held in conjunction with the 8th International Symposium on High-Performance Computer Architecture*, Cambridge, MA, USA, February 2002.

[14] Julia Chen, Philo Juang, Kevin Ko, Gilberto Contreras, David Penry, Ram Rangan, Adam Stoler, Li-Shiuan Peh, and Margaret Martonosi. Hardware-modulated parallelism in chip multiprocessors. *SIGARCH Comput. Archit. News*, 33(4):54–63, 2005.

[15] K. Cho, K. Mitsuya, and A. Kato. Traffic data repository at the WIDE project. In *USENIX 2000 Annual Technical Conference: FREENIX Track*, pages 263–270, San Diego, CA, June 2000.

[16] K. G. Coffman and A. M. Odlyzko. Internet growth: is there a "Moore's law" for data traffic? pages 47–93, 2002.

[17] Michele Colajanni and Mirco Marchetti. A parallel architecture for stateful intrusion detection in high traffic networks. In *IEEE / IST Workshop on Monitoring, Attack Detection and Mitigation*, Tuebingen, Germany, September 2006.

[18] Network Security: 128-core processor is designed for secure LAN. *http://www.eetimes.com*.

[19] Theofanis Constantinou, Yiannakis Sazeides, Pierre Michaud, Damien Fetis, and Andre Seznec. Performance implications of single thread migration on a chip multi-core. *SIGARCH Comput. Archit. News*, 33(4):80–91, 2005.

[20] G. Coretez. Fun with packets: Designing a stick; draft white paper on stick. *http://www.eurocompton.net/stick/*.

[21] P. Crowley, M. A. Franklin, H. Hadimioglu, and P. Z. Onufryk. *Network Processor Design: Issues and Practices*, volume 1, chapter Network Processors: An Introduction to Design Issues. Morgan Kaufmann Publishers, San Mateo, CA, USA, 2002.

[22] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stam, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, September 1997.

[23] N. Forbes and M. Foster. The end of Moore's law? *Computing in Science & Engineering*, 5(1):18–19, Jan/Feb 2003.

[24] M. Franklin and S. Datar. Pipeline task scheduling on network processors. In *Proc. of 3rd Workshop on Network Processors - NP3, Held in conjunction with the 10th International Symposium on High-Performance Computer Architecture*, Madrid, Spain, February 2004.

[25] George Gilder. Telecosm. Free Press, September 2000.

[26] R. Golla. Niagara2: A highly threaded Server-on-a-Chip. *http://www.opensparc.net/pubs/preszo/06/04-Sun-Golla.pdf*, 2006.

[27] Bart Haagdorens, Tim Vermeiren, and Marnix Goossens. Improving the performance of signature-based network intrusion detection sensors by multi-threading. In *Information Security Applications, 5th International Workshop, WISA, Jeju Island, Korea, Revised Selected Papers*, volume 3325 of *Lecture Notes in Computer Science*, pages 188–203. Springer, August 2004.

[28] Christopher L. Hayes and Yan Luo. DPICO: a high speed deep packet inspection engine using compact finite automata. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 195–203, New York, NY, USA, 2007. ACM.

[29] Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio Nishimura, Yoshimori Nakase, and Teiji Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. *Proceedings of the 19th Annual Intl. Symposium on Computer Architecture*, pages 136–145, 1992.

[30] J. Won-Ki Hong. Invited Talk: Internet Traffic Monitoring and Analysis using NG-MON. In *IEEE 6th International Conference on Advanced Communication Technology (ICACT-6)*, Phoenix Park, Republic of Korea, February 2004.

[31] IBM PowerNP Family. *http://www.research.ibm.com/journal/rd/472/allen.html*.

[32] Intel Pentium III processor: Developers manual. In Technical Report, 1999.

[33] Intel Pentium II processor - datasheets. In Technical Report, 2002.

[34] Intel IXP Network Processor Family. *http://www.intel.com/design/network/products/npfamily/.*

[35] Van Jacobson. A new way to look at networking. *http://video.google.com*, August 2006.

[36] Daniel A. Jiménez and Calvin Lin. Neural methods for dynamic branch prediction. *ACM Transactions on Computer Systems*, 20(4):369–397, November 2002.

[37] NPU startup Kayamba packs 256 threads. *http://www.eetimes.com*.

[38] Sunil Kim and Jun yong Lee. A system architecture for high-speed deep packet inspection in signature-based network intrusion prevention. *J. Syst. Archit.*, 53(5-6):310–320, 2007.

[39] E. Kohler, J. Li, V. Paxson, and S. Shenker. Observed structure of addresses in IP traffic. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment workshop*, pages 253–266, Pittsburgh, PA, USA, August 2002.

[40] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful intrusion detection for high-speed networks. In *Proc. IEEE Symposium Security and Privacy, IEEE Computer Society Press*, CA, USA, 2002.

[41] Christopher Kruegel, Fredrik Valeur, Giovanni Vigna, and Richard Kemmerer. Stateful intrusion detection for high-speed networks. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 285, Washington, DC, USA, 2002. IEEE Computer Society.

[42] J. T. Kuehn and B. J. Smith. The horizon supercomputing system: architecture and software. In *Supercomputing '88: Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, pages 28–34, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[43] J. Larus and R. Rajwar. *Transactional Memory*. Morgan Claypool, 2006.

[44] B. K. Lee and L. K. John. NpBench: A Benchmark Suite for Control Plane and Data Plane Applications for Network Processors. In *IEEE International Conference on Computer Design: VLSI in Computers & Processors (ICCD)*, pages 226–233, San Jose, CA, USA, October 2003.

[45] Barry M. Leiner, Vinton G. Cerf, David D. Clark, Robert E. Kahn, Leonard Kleinrock, Daniel C. Lynch, Jon Postel, Larry G. Roberts, and Stephen Wolf. A brief history of the Internet. Retrieved 8/8/02 from *http://www.isoc.org/internet/history/brief.shtml*, 2000.

[46] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Trans. Netw.*, 2(1):1–15, 1994.

[47] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart memories: a modular reconfigurable architecture. pages 161–171, 2000.

[48] Arindam Mallik, Yu Zhang, and Gokhan Memik. Automated task distribution in multicore network processors using statistical analysis. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 67–76, New York, NY, USA, 2007. ACM.

[49] Pedro Marcuello, Antonio González, and Jordi Tubella. Speculative multithreaded processors. In *ICS '98: Proceedings of the 12th international conference on Supercomputing*, pages 77–84, New York, NY, USA, 1998. ACM.

[50] Michael R. Marty and Mark D. Hill. Virtual hierarchies. *IEEE Micro*, 28(1):99–109, 2008.

[51] Scott McFarling. Combining branch predictors. Technical Report TN-36, Compaq Western Research Lab., June 1993.

[52] James D. Meindl. Beyond Moore's Law: The Interconnect Era. *Computing in Science and Engg.*, 5(1):20–24, 2003.

[53] S. Melvin, M. Nemirovsky, E. Musoll, J. Huynh, R. Milito, H. Urdaneta, and K. Saraf. A massively multithreaded packet processor. In *Proc. of 2nd Workshop on Network Processors - NP2, Held in conjunction with the 9th International Symposium on High-Performance Computer Architecture*, pages 64–74, Anaheim, CA, USA, February 2003.

[54] G. Memik, W. H. Mangione-Smith, and W. Hu. Netbench: A benchmarking suite for network processors. In *IEEE International Conference Computer-Aided Design (ICCAD)*, San Jose, CA, USA, November 2001.

[55] G. Minshall. TCPdpriv: Program for Eliminating Confidential Information from Traces. Ipsilon Networks, Inc. http://ita.ee.lbl.gov/html/contrib/tcpdpriv.html.

[56] Gordon E. Moore. Cramming more components onto integrated circuits. *Readings in computer architecture*, pages 56–59, 2000.

[57] Jose Eduardo Moreira. *On the implementation and effectiveness of autoscheduling for shared-memory multiprocessors*. PhD thesis, Champaign, IL, USA, 1995.

[58] A. Nemirovsky. Towards characterizing network processors: Needs and challenges. *Xstream Logic Inc., white paper*, November 2000.

[59] National Lab of Applied Network Research. *http://pma.nlanr.net/Traces*.

[60] Network Processing Forum. *http://www.npforum.org*.

[61] A. M. Odlyzko. Internet traffic growth: Sources and implications. In *Optical Transmission Systems and Equipment for WDM Networking II, B. B. Dingel, W. Weiershausen, A. K. Dutta, and K.-I. Sato, eds., Proc. SPIE, vol. 5247*, pages 1–15, September 2003.

[62] R. Pang and V. Paxson. A high-level programming environment for packet trace anonymization and transformation. In *Proceedings of the ACM SIGCOMM Conference*, Karlsruhe, Germany, August 2003.

[63] Kihong Park, Gitae Kim, and Mark Crovella. On the relationship between file sizes, transport protocols, and self-similar network traffic. In *Procs. of the 1996 International Conference on Network Protocols (ICNP '96)*, page 171, Washington, DC, USA, 1996. IEEE Computer Society.

[64] V. Paxson, R. Sommer, and N. Weaver. An architecture for exploiting multi-core processors to parallelize network intrusion prevention. In *Proc. IEEE Sarnoff Symposium*, May 2007.

[65] William Plishker. Automated task allocation for network processors, October 2004.

[66] Jr. R. H. Halstead and T. Fujita. MASA: a multithreaded processor architecture for parallel symbolic computing. *SIGARCH Comput. Archit. News*, 16(2):443–451, 1988.

[67] Raza Microelectronics Inc. *http://www.razamicroelectronics.com*.

[68] RedIRIS: Spanish National Research Network. *http://www.rediris.es*.

[69] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.

[70] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 13th Conference on Systems Administration (LISA-99)*, pages 229–238, Seattle, WA, USA, November 1999. USENIX.

[71] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. Trace processors. In *Procs. of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 138–148, Washington, DC, USA, 1997. IEEE Computer Society.

[72] P. Sagmeister, G. Dittmann, A. Herkersdorf, and D. Webb. "methodology for testing high-speed network devices with predicted traffic". In *IEEE Gigabit Networking Workshop (GBN)*, Anchorage, Alaska, April 2001.

[73] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[74] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. *SIGARCH Comput. Archit. News*, 31(2):422–433, 2003.

[75] L. Schaelicke, T. Slabach, B. Moore, and C. Freeland. Characterizing the performance of network intrusion detection sensors. In *Procs. of the 6th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Pittsburgh, PA, USA, September 2003.

[76] Derek L. Schuff, Yung Ryn Choe, and Vijay S. Pai. Conservative vs. optimistic parallelization of stateful network intrusion detection. In *Procs. of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 138–139, 2007.

[77] Derek L. Schuff and Vijay S. Pai. Design alternatives for a high-performance self-securing ethernet network interface. In *Procs. of 21th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10. IEEE, March 2007.

[78] M. J. Serrano, R. Wood, and M. Nemirovsky. A study on multistreamed superscalar processors. Technical Report 93-05, University of California Santa Barbara, 1993.

[79] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[80] B. J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. In *Real Time Signal Processing IV, Proceedings of SPIE*, 1981.

[81] The open source network intrusion detection system. *http://www.snort.org*.

[82] Snot v0.92 alpha. *http://www.stolenshoes.net/sniph/snot-0.92a-README.txt*.

[83] Anand Srinivasan, Philip Holman, James Anderson, Sanjoy Baruah, and Jasleen Kaur. Multiprocessor scheduling in processor-based router platforms: Issues and ideas. In *Network Processor Design: Issues and Practices*, November 2003.

[84] A. Srivastava and A. Eustace. ATOM - A system for building customized program analysis tools. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 196–205, June 1994.

[85] S. N. Storino, A. G. Aipperspach, J. M. Borkenhagen, R. J. Eickemeyer, S. R. Kunkel, S. B. Levenstein, and G. J. Uhlmann. A Commercial Multithreaded RISC Processor. In *Digest of Papers, International Solid-State Circuits Conference*, pages 236–237, San Francisco, CA, February 1998.

[86] Sun Microsystems. UltraSPARC T1 Supplement to the UltraSPARC Architecture 2005, 2006.

[87] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *Procs. of the 31st annual International Symposium on Computer Architecture*, page 2, Munchen, Germany, 2004.

[88] Marc Tremblay, Jeffrey Chan, Shailender Chaudhry, Andrew W. Conigliaro, and Shing Sheung Tse. The MAJC Architecture: A Synthesis of Parallelism and Scalability. *IEEE Micro*, 20(6):12–25, 2000.

[89] Dean M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *Procs. of 22nd Annual Computer Measurement Group Conference*, pages 819–828, December 1996.

[90] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual Intl. Symposium on Computer Architecture*, pages 191–202, May 1996.

[91] Matthias Vallentin, Robin Sommer, Jason Lee, Craig Leres, Vern Paxson, and Brian Tierney. The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In *Procs. of 10th International Symposium Recent Advances in Intrusion Detection (RAID)*, pages 107–126, September 2007.

[92] J. Verdú, J. García, M. Nemirovsky, and M. Valero. Analysis of traffic traces for stateful applications. In *Proc. of 3rd Workshop on Network Processors - NP3, Held in conjunction with the 10th International Symposium on High-Performance Computer Architecture*, pages 120–124, Madrid, Spain, Feb 2004.

[93] J. Verdú, J. García, M. Nemirovsky, and M. Valero. Traffic aggregation impact. In *Proc. of 5th Workshop on Memory Performance: Dealing with Applications, systems and architecture - MEDEA5, Held in conjunction with the International Conference on Parallel Architectures and Compilation Techniques (PACT-2004)*, Antibes Juan-les-Pins, France, September 2004.

[94] Tim Vermeiren, Eric Borghs, and Bart Haagdorens. Evaluation of software techniques for parallel packet processing on multi-core processors. In *Proceedings of IEEE Consumer Communications & Networking Conference (CCNC)*, Las Vegas, NV, USA, January 2004.

[95] Lucian N. Vintan and Mihaela Iridon. Towards a high performance neural branch predictor. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2, pages 868–873, July 1999.

[96] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 26, pages 176–189, New York, NY, 1991. ACM Press.

[97] N. Weng and T. Wolf. Pipelining vs. multiprocessors - choosing the right network processor system topology. In *Procs. of Advanced Networking and Communications Hardware Workshop (ANCHOR) in conjunction with ISCA*, Munich, Germany, June 2004.

[98] N. Weng and T. Wolf. Profiling and mapping of parallel workloads on network processors. In *Proc. of The 20th Annual ACM Symposium on Applied Computing (SAC)*, pages 890–896, Santa Fe, NM, March 2005.

[99] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.

[100] T. Wolf and Mark A. Franklin. Commbench - a telecommunications benchmark for network processors. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 154–162, Austin, TX, USA, April 2000.

[101] T. Wolf, P. Pappu, and M. A. Franklin. Predictive scheduling of network processors. *Computer Networks*, 41(5):601–621, April 2003.

[102] J. Xu, J. Fan, M. Ammar, and S. B. Moon. On the design and performance of prefix-preserving ip traffic trace anonymization. In *IMW '01: Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 263–266, New York, NY, USA, 2001. ACM.

[103] W. Yamamoto and M. Nemirovsky. Increasing superscalar performance through multistreaming. In *Proceedings of the Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 49–58, Sept 1995.

[104] W. Yamamoto, M. Serrano, A. Talcott, R. Wood, and M. Nemirovsky. Performance estimation of multistreamed, supersealar processors. In *Procs. of 27th Annual Hawaii International Conference on System Sciences (HICSS)*, pages 195–204, Maui, Hawaii, January 1994. IEEE Computer Society.

[105] Donald Yeung, John Kubiatowicz, and Anant Agarwal. MGS: a multigrain shared memory system. *SIGARCH Comput. Archit. News*, 24(2):44–55, 1996.

[106] Fang Yu. *High Speed Deep Packet Inspection with Hardware Support*. PhD thesis, EECS Department, University of California, Berkeley, Nov 2006.