View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by D-Scholarship@Pitt

VIRTUALIZATION OF NETWORK I/O ON MODERN OPERATING SYSTEMS

by

Takashi Okumura

B.A., Keio University, Japan, 1996

M.A., Keio University, Japan, 1998

M.S., University of Pittsburgh, 2000

M.D., Asahikawa Medical College, 2007

Submitted to the Graduate Faculty of
the Arts and Science in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2007

UNIVERSITY OF PITTSBURGH

DEPARTMENT OF COMPUETR SCIENCE

This dissertation was presented

by

Takashi Okumura

It was defended on

August 24th, 2006

and approved by

Dr. Daniel Mossé

Dr. Ahmed Amer

Dr. Bruce R. Childers

Dr. Hideyuki Tokuda, Keio University

Dissertation Director: Dr. Daniel Mossé

Copyright © by Takashi Okumura
2007

VIRTUALIZATION OF NETWORK I/O ON MODERN OPERATING SYSTEMS

Takashi Okumura, Ph.D

University of Pittsburgh, 2007

Network I/O of modern operating systems is incomplete. In this network age, users and their applications are still unable to control their own traffic, even on their local host. Network I/O is a shared resource of a host machine, and traditionally, to address problems with a shared resource, system research has virtualized the resource. Therefore, it is reasonable to ask if the virtualization can provide solutions to problems in network I/O of modern operating systems, in the same way as the other components of computer systems, such as memory and CPU.

With the aim of establishing the virtualization of network I/O as a design principle of operating systems, this dissertation first presents a virtualization model, *hierarchical virtualization of network interface*. Systematic evaluation illustrates that the virtualization model possesses desirable properties for virtualization of network I/O, namely flexible control granularity, resource protection, partitioning of resource consumption, proper access control and generality as a control model. The implemented prototype exhibits practical performance with expected functionality, and allowed flexible and dynamic network control by users and applications, unlike existing systems designed solely for system administrators.

However, because the implementation was hardcoded in kernel source code, the prototype was not perfect in its functional coverage and flexibility. Accordingly, this dissertation investigated how to decouple OS kernels and packet processing code through virtualization, and studied three degrees of code virtualization, namely, limited virtualization, partial virtualization, and complete virtualization. In this process, a novel programming model was

presented, based on embedded Java technology, and the prototype implementation exhibited the following characteristics, which are desirable for network code virtualization. First, users program in Java to carry out safe and simple programming for packet processing. Second, anyone, even untrusted applications, can perform injection of packet processing code in the kernel, due to isolation of code execution. Third, the prototype implementation empirically proved that such a virtualization does not jeopardize system performance.

These cases illustrate advantages of virtualization, and suggest that the hierarchical virtualization of network interfaces can be an effective solution to problems in network I/O of modern operating systems, both in the control model and in implementation.

TABLE OF CONTENTS

I INTRODUCTION . . . 1
- I.1 Network I/O and Virtualization . . . 1
- I.2 Functional disorder in network I/O control services . . . 2
- I.3 Structural disorder in network I/O implementation framework . . . 4
- I.4 Summary - "Network I/O needs virtualization" . . . 6
- I.5 Nomenclature - virtualization . . . 7

II RELATED WORK . . . 9
- II.1 Classification of network I/O services . . . 9
- II.2 Functional analysis of network I/O services . . . 12
  - II.2.1 Class-A Services: Traffic Control . . . 12
  - II.2.2 Class-B Services: Security Control . . . 13
  - II.2.3 Class-C Services: Resource Management . . . 15
  - II.2.4 Class-D Services: Access Control . . . 16
  - II.2.5 Limitation of the approaches . . . 16
- II.3 Structural analysis of network I/O implementation framework . . . 18
  - II.3.1 Processing location . . . 19
    - II.3.1.1 Userland protocol processing . . . 19
    - II.3.1.2 Kernel extensibility . . . 20
  - II.3.2 Degree of virtualization . . . 21
    - II.3.2.1 Packet filters . . . 22
    - II.3.2.2 Structured approach in protocol processing . . . 22
    - II.3.2.3 In-kernel execution of packet handling code . . . 23
    - II.3.2.4 Kernel extensibility revisited . . . 24
- II.4 Summary - "Network I/O has been poorly virtualized" . . . 25

III VIRTUALIZATION MODEL OF NETWORK I/O . . . 27

III.1 Abstraction for network I/O virtualization 27
III.1.1 Abstraction of end-to-end communication 28
III.1.1.1 Low-level Abstraction 28
III.1.1.2 High-level Abstraction 29
III.1.2 Virtualization inside end-host system 30
III.1.2.1 Horizontal virtualization of the network interface . 30
III.1.2.2 Hierarchical virtualization of the network interface 31
III.2 Hierarchical virtualization of network interfaces 33
III.2.1 Generality of Control 33
III.2.2 Rules for Resource Protection 34
III.2.3 Access control and system interface 36
III.3 Qualitative evaluation as a virtualization model 37
III.4 Summary - "Hierarchical virtualization is the one" 39
IV PROTOTYPE IMPLEMENTATION OF THE MODEL 40
IV.1 Prototype Implementation 40
IV.1.1 Implementation overview 40
IV.1.1.1 Packet Hook 41
IV.1.1.2 Packet Scheduling 42
IV.1.1.3 Packet Marking 45
IV.1.1.4 Tuning for better performance 46
IV.1.2 System Interface 46
IV.1.2.1 Sample Scenario 47
IV.1.2.2 File System Abstraction 48
IV.1.2.3 Operation Model 49
IV.1.2.4 Remarks 51
IV.1.3 Sample Applications 51
IV.1.3.1 End-User commands for traffic control 52
IV.1.3.2 Control of commodity servers 52
IV.1.3.3 QoS manager 53
IV.2 Extension of Functionality - packet monitor 55

IV.2.1 Overview - the Port mechanism 55
IV.2.2 Implementation detail 56
IV.2.3 Implication of the VIF monitoring capability 58
IV.2.3.1 Monitoring Granularity 58
IV.2.3.2 Integration of Monitoring and Control 59
IV.2.3.3 Possible applications 59
IV.3 Evaluation 60
IV.3.1 Traffic Profile 61
IV.3.2 Overhead of management operations 62
IV.3.3 Scheduler Overhead 62
IV.3.4 Port mechanism 65
IV.3.4.1 Capturing performance 66
IV.3.4.2 Impact on Throughput of mainstream traffic . . . 67
IV.3.4.3 Packet Diverting 68
IV.3.5 Qualitative evaluation 69
IV.3.5.1 Kernel implementation level 69
IV.3.5.2 Functionality level 70
IV.3.5.3 Application level 71
IV.4 Summary - “Hierarchical virtualization can be efficient” 73
V LIMITED VIRTUALIZATION OF PACKET PROCESSING CODE 74
V.1 The filter mechanism - Overview 74
V.2 Implementation 76
V.3 Evaluation 78
V.3.1 Quantitative evaluation 78
V.3.1.1 Translation overhead 78
V.3.1.2 Filtering performance - microscopic 79
V.3.1.3 Filtering performance - macroscopic 80
V.3.1.4 Discussion 80
V.3.2 Qualitative evaluation 81
V.3.2.1 Achievements 82

V.3.2.2 Limitation 82
V.3.2.3 Implication as a security mechanism 83
V.4 Summary - “It is limited” 84
VI **PARTIAL VIRTUALIZATION OF PACKET PROCESSING CODE** 86
VI.1 In-kernel Virtual Machine for packet processing 86
VI.1.1 VIFlet model - Overview 87
VI.1.2 Execution environment - a lightweight in-kernel JVM/JIT . 88
VI.1.2.1 NVM - A lightweight Java Virtual Machine 89
VI.1.2.2 NYA - A Just-In-Time compiler 89
VI.1.3 Lessons learned 90
VI.1.3.1 Coding process 90
VI.1.3.2 Porting JVM into kernel space 91
VI.1.3.3 Customization for performance boosting 92
VI.1.3.4 Reducing cost for runtime checking 93
VI.1.3.5 Garbage collection and protection mechanism . . 93
VI.2 Quantitative evaluation 94
VI.2.1 NVM/NYA Module Performance 94
VI.2.1.1 Overall Performance 94
VI.2.1.2 Performance Breakdown 96
VI.2.1.3 Compilation cost 98
VI.2.2 VIFlet system performance 99
VI.2.2.1 A case study: priority queuing 99
VI.2.2.2 A simple VIFlet performance 100
VI.2.3 Breakdown of a packet processing 101
VI.3 Qualitative evaluation 103
VI.3.1 Achievements 103
VI.3.1.1 Kernel extensibility for packet processing 103
VI.3.1.2 Computation model 104
VI.3.1.3 In-kernel Java VM for packet processing 105
VI.3.2 Limitations 106

VI.3.2.1 Computational model 106
VI.3.2.2 Protection 107
VI.3.2.3 Performance 108
VI.3.3 Special remark on in-kernel Java VM projects 109
VI.4 Summary - "This is promising" 110
VII TOWARD COMPLETE VIRTUALIZATION OF NETWORK I/O 112
VII.1 Extension of VIFlet model for complete virtualization 112
VII.1.1 Protocol stack implementation and Capabilities 113
VII.1.2 Capabilities and the VIFlet model 115
VII.1.2.1 Complication of Write access in the model 116
VII.1.2.2 Complication of Creation capability in the model 117
VII.1.2.3 Externality and the VIFlet model 118
VII.1.3 Suggested Design 119
VII.2 Implementation approach for further performance gain 120
VII.2.1 Research Trend 120
VII.2.2 Components of packet processing and possible approaches . 122
VII.2.2.1 Virtualization Ratio 122
VII.2.2.2 Instruction Set Architecture 123
VII.2.2.3 Language and Compiler 123
VII.2.2.4 Profiling and Optimization 124
VII.2.3 Suggested Design 125
VII.3 Contribution and Significance 125
VII.3.1 Classification of virtualization technology 126
VII.3.2 Hierarchical virtualizations 127
VII.3.2.1 Hierarchical CPU scheduler 128
VII.3.2.2 Hierarchical resource specification 130
VII.3.2.3 Hierarchical modeling of protocol processing . . . 130
VII.3.3 Hierarchical virtualization of network interface 131
VII.4 Summary - "Virtualization is advantageous" 132
VIII CONCLUSIONS 134

VIII.1 Conclusion 134
VIII.2 Contributions 135
VIII.2.1 Resource management model of network I/O 135
VIII.2.2 VIFlet model for extensibility of network I/O code 136
VIII.2.3 Prototype implementation and its profile 137
VIII.2.4 Analysis of network I/O on modern operating systems . . . 138
VIII.3 Future work 138
VIII.3.1 Complete virtualization 139
VIII.3.2 Integration with task scheduler 139
VIII.3.3 End-to-end QoS 140
VIII.4 Closing remarks 141
GLOSSARY 143
BIBLIOGRAPHY 146

LIST OF TABLES

4.1 System Call Profile (in μsecs). 63
6.1 Optimization Options 90
6.2 Compilation cost sample 98
7.1 Classification of virtualization technology and examples 126

LIST OF FIGURES

2.1 Functionality-based classification of network control services 10

2.2 Class-A mechanisms 13

2.3 Class-B mechanisms 14

2.4 Class-C mechanisms 15

2.5 Class-D mechanisms 16

3.1 Horizontal virtualization and potential conflicts 31

3.2 Hierarchical virtual network interface model 32

3.3 Various virtual network interfaces 33

3.4 Various virtual network interfaces 34

3.5 Resource protection and Access control model 35

3.6 Sandboxing a virtual network interface 36

3.7 Sample scenario 38

4.1 Implementation Overview 41

4.2 The hierarchical packet scheduler 43

4.3 Sample configuration of a hierarchical virtual network interface structure . . 47

4.4 Directory structure for the sample VIF configuration in Figure 4.3. 48

4.5 Simple script for fair-sharing of wireless network bandwidth 54

4.6 Port interface of the VIF system 55

4.7 Header Format of the packet filter 57

4.8 Experimental Setting 61

4.9 Throughput of VIFs for both input (VIF 4) and output (VIFs 1-3, 5) traffic . 62

4.10 Scheduler overhead 64

4.11 Experimental Setting 66
4.12 Packet Capturing Performance of BPF and the port 67
4.13 End-to-End throughput while being captured 67
4.14 End-to-End throughput of Packet Diverting 68
4.15 Functional Coverage of the prototype implementation 70
5.1 Code generation and Instrumentation of filter binary in the kernel 75
5.2 Generation and invocation of binary filter code 76
5.3 Translation Overhead 78
5.4 Packet Filtering Performance of IPFW and the filter in CPU cycles 79
5.5 End-to-End throughput with attack traffic 80
6.1 VIFlet model 87
6.2 Sample VIFlet code 88
6.3 Overview of the coding process 91
6.4 Sieve Benchmark 95
6.5 Object Sort Benchmark 96
6.6 Benchmark of primitive Java bytecode operations 97
6.7 Experimental Setting 99
6.8 A case of priority queuing VIFlet 100
6.9 A simple VIFlet Performance 101
6.10 Breakdown of processing for an outgoing packet (in μsec) 102
7.1 Packet processing locations and their properties 113
7.2 Imaginary overview of extended VIFlet model 115
7.3 Write access and its complication 116
7.4 Creation capability and its complication 117
7.5 Two possible designs to assure consistency of protocol code 119
7.6 Flexible virtualization by cache servers 128
7.7 Hierarchical CPU scheduler 129
7.8 Hierarchical packet scheduler 129
7.9 Class Based Queuing and hierarchical specification 130
7.10 x-Kernel and its resource management 131

ACKNOWLEDGMENTS

I would first like to thank my supervisor for giving me the opportunity to pursue research career in this field, and for his cordial support. Without his guide and assistance, it would have been impossible to finish this long journey. I would also like to thank the following professors at University of Pittsburgh, Mark Moir, Manas Saksena, Bruce Childers, and Ahmed Amer, who joined my M.S. and Ph.D. committees, and kindly coauthored the research outcomes. My special thanks are also due to Professor Hideyuki Tokuda at Keio University, who participated in the dissertation committee.

I am greatly indebted to Akira Kato, Sho Fujita, Hiroshi Esaki, Osamu Nakamura, Minami Masaki, and Jun Murai, members of WIDE project, a research consortium of internetworking technology in Japan. They supported my research activities at Asahikawa Medical College, Japan, where I studied toward my M.D. degree, my other source of identity. My pursuit of a Ph.D. degree in the United States and an M.D. degree in Japan was also supported by the following professors at Asahikawa Medical College, Takashi Sakamoto, Masaharu Takahashi, Hiroshi Suzuki, Hiroyuki Hirokawa, Yusuke Saito, Masakazu Haneda, and Takahiko Yoshida. Tomoya Okazaki kindly supported to start up and maintain a research environment at Information Processing Center of the college. I also appreciate all the help from my colleagues, members of the Netnice.org project and the Netnice working group of WIDE project. They helped me in many aspects through their participation in these groups. James Larkby-Lahet deserves special thanks for his help in the proofreading.

The research has been financially supported in part by NSF under grants 0087609, 0121658, and 0325353. It was also partly supported by the Telecommunications Advancement Organization of Japan (TAO), under grant JGN-P122518, and Information-technology Promotion Agency (IPA). For the IPA projects, I especially thank Ikuo Takeuchi at University of Tokyo and Kazuhiro Kato at University of Tsukuba for their commitments.

I INTRODUCTION

I.1 NETWORK I/O AND VIRTUALIZATION

Network I/O of modern operating systems is incomplete. A program may easily exhaust link bandwidth and the entire buffer pool, since current operating systems do not have mechanisms to arbitrate network resource consumption among concurrent connections. Traffic control mechanisms are provided on some operating systems, but only for system administrators, since they are accessible only through privileged instructions. Consequently, in this network age, network users and their applications are still unable to control their own traffic, even on their local host, let alone in the network.

Such Network I/O is a shared resource of a host machine. Traditionally, to address problems with a shared resource, system research has *virtualized* the resource. Many programs can be run on a machine without interfering with each other, because the operating system virtualizes the physical memory into partitioned domains, thereby protecting execution of each code from illegal memory accesses. Virtualization is also the concept behind partitioning CPU cycles among concurrent processes, allowing far better system utilization while guaranteeing appropriate response time to interactive sessions. As these cases illustrate, virtualization has been a legitimate approach for resource sharing problems on computer systems. Indeed, it is a key concept of system research, which forms the basis for almost all the work in the field of study.

Therefore, the logical next question to pose is : *can virtualization provide a solution to problems in network I/O of modern operating systems?*, in the same way as it has for other components of computer systems. With hindsight, it seems almost historically inevitable that components of modern computer systems became virtualized. We may show that it

would also be historically inevitable that network I/O of modern operating systems becomes virtualized. This dissertation investigates this thesis and aims at empirically establishing the virtualization of network I/O as a design principle of operating systems.

However, in this problem domain, most research efforts and implementation attempts have been oriented toward ad-hoc solutions, as illustrated shortly. Accordingly, it is unclear even what the fundamental problems in network I/O of modern operating systems are. Even when research attempted virtualization of network I/O, most of the proposals insufficiently virtualized network interfaces or protocol processing, without systematic investigation of the nature of network I/O. Consequently, this introduction first illustrates incompleteness of network I/O of modern operating systems, in its control and implementation, to further clarify focus of the dissertation.

I.2 FUNCTIONAL DISORDER IN NETWORK I/O CONTROL SERVICES

Developers' communities have been making enormous efforts in network I/O services, to meet a variety of operational needs. For example, to meet the needs for queuing control and traffic shaping, the BSD community developed several mechanisms for advanced queuing management, such as ALTQ [31] and Dummynet [113], while the Linux community developed similar mechanisms [65, 37]. As solutions to network security problems, the BSD community created the Packet Filter (PF) [64] for packet filtering, whereas the Linux community built Netfilter mechanisms [73]. The Windows OS also has similar mechanisms [96].

These mechanisms for network I/O control are widely used nowadays, for the following reasons. First, Unix workstations with multiple network interfaces have been used as routers, and there are great demands for router-like control capability on these hosts. Second, network-borne threats have urged end-host systems to possess some mechanisms to protect themselves, such as packet filters. Because of this, almost every operating system is equipped with network security mechanism, nowadays. Third, the processor performance of end systems has increased to the point where a single node can easily saturate attached network resources. This growth necessitates efficient means to manage their resource consumption

within that host, particularly, on a shared media, such as wireless network, where traffic control can be achieved only by the cooperation of end-nodes. Lastly, quality-conscious network applications have come into wide use, such as voice streaming and packet video. They require services for traffic control, such as the ability to reserve bandwidth on the path, for proper communication quality.

However, because the mechanisms have been developed mostly as ad-hoc solutions to problems confronting each community, there are many problems left untouched in these mechanisms, as exemplified below.

1. The implementations are designed as administrative tools, using privileged instructions that only administrators can use. This was a reasonable choice for system administrators, because they are also administrators of the network the machines are connected to, in many institutions. The control model developed for network routers was intuitive for most system administrators and kernel developers. As a result of this design decision, almost all of the implementations cannot be used by users and unprivileged user-level applications.
2. Because the tools are designed for static configuration, as is the case on unmanned routers where an administrator statically sets control parameters at system configuration time, most mechanisms are not used for dynamic control of network traffic.
3. The resource management viewpoint inherent to operating systems is completely missing in these mechanisms, even though it is indispensable for systems to be functional. A misbehaving program can easily exhaust the entire buffer pool. A low-priority process can easily saturate network bandwidth, starving high-priority processes.
4. There is no compatibility among these implementations, because these implementations were developed independently to meet the needs of independent user communities for each operating system. Consequently, network applications cannot utilize the network control services in an OS-independent manner, and the lack of compatibility formed impenetrable boundaries among operating systems, which substantially inflated development and deployment cost of new technology.

It is clear that programmability is a critical catalyst for growth of computer systems.

For example, the Internet has evolved with the development and general accessibility of a variety of attractive applications for users: email, World Wide Web, file sharing software, instant messenger and IP telephony, just to name a few. Accordingly, the paternalistic model, where only an administrator controls flow of packets in the middle of communication, severely retarded development of new technology in the domain. Indeed, what progress have we made for the network control technology in the last decade? There is a striking contrast with the Web technology, which achieved the Applet, Cascading Style Sheets (CSS), Ajax (Asynchronous JavaScript plus XML), Flash, and even in-lined streaming and video images, starting from the primitive HTML documents. There is a functional disorder in network I/O of modern operating systems, and we need a breakthrough here.

I.3 STRUCTURAL DISORDER IN NETWORK I/O IMPLEMENTATION FRAMEWORK

The services for network I/O control have been mostly developed by closed communities formed for each operating system, and these communities evolved the mechanisms by extending their operating systems each time they found a new problem in the field. In the development process, they *hardcoded* their network I/O code inside the operating systems, and this development style had serious consequences on network technology. First, the hardcoding tightly associated each implementation with operating system kernels, and inflated the porting cost of the packet processing code. Second, it made the implementations quite inflexible, and inflated the cost to install new programs onto working systems (can a novice user easily apply kernel patches?). Third, it made bundled mechanisms on each OS serve as de-facto standards for the platforms. They caused it to be far easier to simply exploit existing mechanisms, than to invent novel mechanisms.

Through their problem-oriented development process, OS communities successfully developed handy and qualified implementations, usable for daily operations by system administrators. At the same time, however, this prevented technical innovation, and has left the problems in network I/O of modern operating systems unexplored for years.

Such a difficulty on modern operating systems has been addressed, in the research field, as kernel extensibility studies. Indeed, operating system researchers have been working to detach network I/O code from the underlying operating system, and have proposed a variety of frameworks with the aim of extending system flexibility to facilitate technical innovation. Sharing the same goal, but with a a contrasting approach, is to allow applications to execute their own packet processing code at userland, which has also been studied in the research field.

However, unfortunately, the proposed schemes for network I/O extensibility did not replace the existing implementations hardcoded in current operating systems, nor provided decisive solution to the problems we had. In reality, any proposal for a novel network protocol or control still needs to be prototyped in an operating system first. Then, it experiences laborious politics in the standardization process, painful porting efforts to major operating systems, and lobbying for stubborn OS gurus to incorporate the developed code into their source repository. Even with all this effort, it only creates the opportunity for deployment, and general acceptance is far from certain.

A clear illustration is IPv6, the next generation Internet Protocol. Evangelists fought for years to build a prototype and to port the protocol to major operating systems, just to be considered as a practical solution in the field. But, this is the most favorable case, in that the attempt has been made by most distinguished hackers in the field and financially supported by governments of economic superpowers. In most academic projects, they are left only to publish several publications, after all the implementation efforts. The output of most open source projects is an academic exercise, unless they are adopted by production operating systems.

As illustrated, there exists a significant gap, almost *a death valley*, between prototyping and deployment, in networking technology. This gap is all caused by the customary hard-coding of packet processing code, and we have another compatibility problem here inside kernels, not just in the application layer, which has considerably disturbed development of network software and proliferation of new technology. This is structural disorder in network I/O implementations, and we need a breakthrough to overcome the difficulty, here again.

I.4 SUMMARY - "NETWORK I/O NEEDS VIRTUALIZATION"

As reviewed thus far, network I/O of modern operating systems is far from complete. The developed control mechanisms for network I/O have similar functionality but no compatibility among them. Existing mechanisms do not provide network control services for system users and network applications, limiting its use to system administrators and their static control commands. Although OS researchers have proposed flexible extensions of kernel functionality, most of the mechanisms for network I/O control are still hardcoded in kernels, limiting flexibility and portability. Accordingly, application programmers and kernel developers are forced to develop independent implementations for each platform whenever they develop a new technology. They cannot even realize priority control of sockets, one of the simplest control, in an OS-independent manner. These "disorders" in the network I/O of current operating systems have substantially inflated development and deployment cost of new networking technology, severely preventing technical innovation.

Traditionally, to address problems of a shared resource, system research has *virtualized* the resource. For example, many programs can be run on a machine without interfering with each other, because the operating system virtualizes the physical memory into partitioned domains. The virtualization technology has also destroyed the OS boundary in code development and deployment. For example, Java technology virtualized code the execution environment as a Java Virtual Machine (JVM), which allowed "Write once, run anywhere" style programming, and realized architecture-neutral executables.

Accordingly, for the problems of network I/O on current operating systems, it is highly probable that *virtualization* can be a decisive solution, as it has been exhibited in other components of computer system. This dissertation investigates the thesis, *virtualization provides a decisive solution to the problems in network I/O of modern operating systems*, in the aim of establishing it as a design principle of operating systems.

This dissertation is organized as follows. Chapter II covers related work and further motivates this dissertation by clarifying incompleteness of existing approaches. Chapter III investigates appropriate abstraction for virtualization of network I/O, contrasting existing models, and presents a novel virtualization scheme of network I/O, *hierarchical virtualization*

of network interface. Chapter IV describes a prototype implementation and evaluates it. The following chapters attempt virtualizing packet processing code, based on the virtualization model, to break the limitations identified in the evaluation. Chapter V presents a simplest case, the virtualized packet classifier code. Chapter VI tries to further advance the virtualization degree of the packet processing environment, to cover packet scheduler processing. Lastly, Chapter VII investigates how complete virtualization of packet processing code is attained. Chapter VIII concludes the dissertation, which investigates whether virtualization can provide appropriate solutions to problems in network I/O of modern operating systems.

I.5 NOMENCLATURE - VIRTUALIZATION

Before starting an in-depth discussion, it would be beneficial to define the term *virtualization* in this document. Confusion arises here, because of the diversity of the term used in system research. For example, researchers virtualized computer hardware, to realize multiplexing of computational resources for better utilization of limited resource [63, 11]. It is also common to use the term to indicate bundling of independent devices to create an illusion of one entity with larger capacity, in the aim of better availability and larger throughput [81].

In this dissertation, *virtualization* is defined as *to modify the underlying details of an entity while keeping its predefined interfaces*. In the case of hardware virtualization, programs can run on the virtualized hardware without any modification, because the virtualized hardware supports the same instruction set architecture, a predefined interface to the executables. In the case of capacity expansion, the requests can be made all the same, as long as the system supports the predefined interface, although the requests are now handled by multiple entities in a totally different manner (e.g., clustered web servers).

Virtualization technology breaks the binding between the exposed service and the underlying mechanism, thereby providing a variety of advantages, such as resource utilization, availability, and throughput. Based on the definition, in this dissertation, we specifically utilize the term virtualization in two contexts: (a) when discussing the control model, virtu-

alization means the abstraction of physical devices; (b) when discussing the implementation of packet processing code, virtualization is used as, for instance, an antonym to “hardcoding”.

The latter usage might be uncommon. But, it is also a common use of the term, indeed, when used in the term of Java Virtual Machine or the Unix Vnode mechanism, which virtualized file system implementation and integrated a variety of file systems into a unified framework in an operating system [77]. Note that these usages meet the definition of the term, virtualization, in that it modifies how services are realized, while keeping interfaces predefined. Appropriateness of the usage will become clearer as the chapters proceed.

Definitions of other terms frequently used in this dissertation are listed in the Glossary.

II RELATED WORK

This chapter clarifies the scope and limitations of existing approaches. To this end, this chapter first presents a classification system of network I/O services (Section II.1), to present existing approaches and to clarify their limitations in the succeeding section (Section II.2). Then, we review different implementations of such services, and identify their technical defects (Section II.3), followed by a brief summary of the entire chapter (Section II.4).

II.1 CLASSIFICATION OF NETWORK I/O SERVICES

Starting from a faithful implementation of the original design, a computing system gradually evolves. In the case of operating systems, file systems have evolved in the face of need for more capacity, lower latency, and better reliability. Memory systems have also changed, starting from a simple memory into hierarchical cache-based organization with virtual address support. The network subsystem, our main interest, has also evolved from faithful implementation of communication protocols to more elaborate ones.

The driving force behind the changes is various needs, such as newer functionality and performance. It is possible to roughly categorize the needs, as shown in Figure 2.1. The first axis (Y-axis in the figure) relates to whether the change adapts to external needs or internal needs. For example, since communication outside the system is abstracted by *flows* and *connections*, we say functionalities that work with these abstractions are *external-needs-oriented*. Conversely, inside the system, we use abstractions that are hidden from outside systems, such as *threads* and *processes*. Accordingly, we say functionalities that work with these abstractions *internal-needs-oriented*. The second axis (X-axis in the figure) relates to

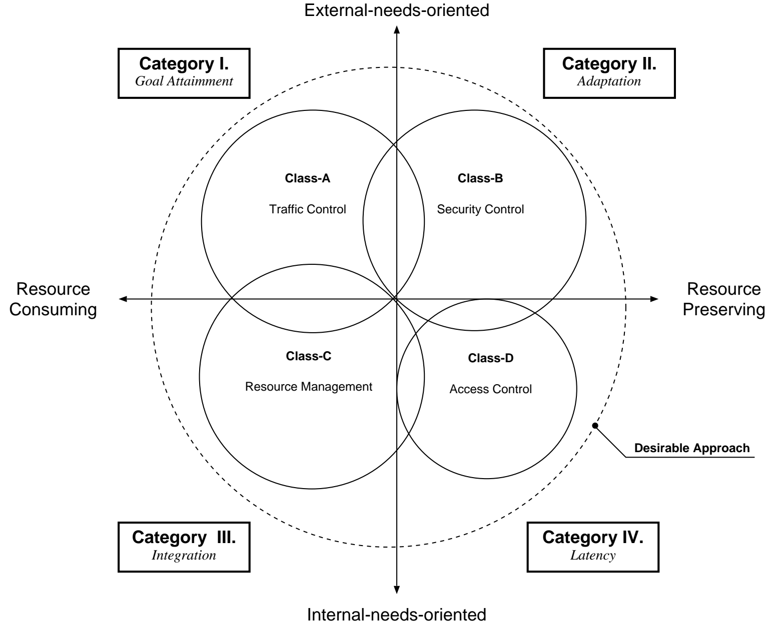


Figure 2.1: Functionality-based classification of network control services

whether the change works to consume resources or to preserve resources. Resource consuming functionality is the set of functions that tries to meet its goal by consuming computational resources, such as CPU cycles and buffers. Resource preserving functionality is the set of functions that tries to preserve resource consumption, typically by protection and limitation. This classification system is similar to that used for *functional analysis* in Sociology, and utilizing the classification system, we can identify four categories as explained below.

- The first category (*Goal-attainment*, in sociological terminology) involves adapting to external needs, by consuming resources of the system. Typical examples include traffic control mechanisms, which consumes CPU cycles on the system to meet an external control goal. We denote these mechanisms as "Class-A" just for convenience.
- The second category (*Adaptation*) involves adapting to external environment, in a way that preserves resources and prohibits unwanted resource consumption. This includes mechanism to prevent wasted CPU cycles, and thus, security mechanisms, such as software firewalls and Denial of Service (DoS) prevention measures are the best examples. As shown, we call them "Class-B". Note that there is an overlap between Class-A and

Class-B, since some mechanisms have both functionalities, QoS and security, as we shall discuss shortly.

- The third category (*Integration*) involves adapting to internal needs, by consuming resources of the system. Examples are resource managers and I/O schedulers, which are designed to ensure proper allocation of computational resources in the system, which are hidden from outside. As indicated, we call them “Class-C”.
- The fourth category (*Latency*, which may sound inappropriate for computer scientists) involves adapting to internal needs, by prohibiting improper use of resources. Internal access control and partitioning mechanism fall in this category, and we call them “Class-D”.

Note that the classification system is designed for *classification of functionality*, not for classification of mechanisms. Since a mechanism may have several functionalities, each mechanism (or a class of mechanisms) does not necessarily have one-to-one relationship to a category.

This classification provides insight into the design of the network subsystem. Under resource constraints, we need to somehow balance the resource consumption and preservation, for the system to be functional. One example is network applications that control the network traffic of the host. An application may throttle the traffic at a network interface (resource preserving), while others are accelerating (resource consuming). It is clear that there is a need for proper arbitration among conflicting requests. Likewise, the system needs to balance the external needs and its internal needs, within the limit of resource constraints. An example is a mechanism to limit the bandwidth of a TCP connection. A network application may control its flow utilizing the mechanism (external-needs oriented), but a user may want the process to avoid exceeding the bandwidth the user granted it (internal-needs oriented). If these conflicting needs are not properly handled, the system status can easily be inconsistent; a low-priority process (with respect to CPU) with greedy network I/O may starve high-priority process with modest network usage. As will be reviewed shortly, existing implementations of network subsystems cannot appropriately handle these cases, suggesting a need for a consistent and complete resource management model.

II.2 FUNCTIONAL ANALYSIS OF NETWORK I/O SERVICES

Based on the classification system in Figure 2.1, a variety of network I/O services are reviewed in this section. Our classification covers a broad spectrum of services proposed on end-host operating systems, providing insight into their nature.

II.2.1 Class-A Services: Traffic Control

The first class is mostly for external-needs-oriented, resource-consuming functionality, which most traffic control mechanisms possess. Almost all of major operating systems nowadays have mechanisms in this class: Linux Advanced Routing and Traffic Control [65], IPFW of FreeBSD [6, 113, 46], ALTQ of NetBSD [31], and PF of OpenBSD [64]. An important property they have in common is that all of these mechanisms use a traffic control model with a "packet classifier". This was developed for control of packets at routers, where packets are typically classified by addresses and port numbers. The traditional control model, with the packet classifier coupled with a packet scheduler for queuing control, is a reasonable approach for control at intermediate routers. However, even for the control on end-host systems, they reused this control model, because Unix workstations with multiple network interfaces have been used as handy routers in early dates. This was also because the flow-oriented model was intuitive for most administrators of networks, who are likely to be administrators of the end-hosts, typically, server systems. We believe that their preference has motivated the field to reuse the router model for end-host network control.

However, a problem arises here. Because end-hosts accommodate various tasks with very diverse profiles, systems need to provide resource management for protection/isolation, which is not required at routers used just by an administrator. For example, it is a reasonable assumption that *resource consumption of a process should not exceed the amount which is granted to the process.* Implementations with the traditional packet classifier model cannot meet such a reasonable requirement on general-purpose operating systems, because the model developed for routers lacks an internal resource management mechanism and protection model. Instead, these mechanisms are protected by privileged system calls, and utilization

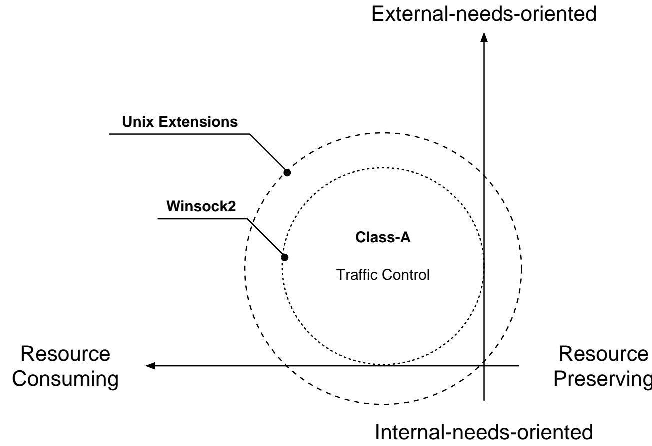


Figure 2.2: Class-A mechanisms

of the mechanisms is strictly limited for special purposes.

The characteristics of the existing approaches are illustrated in Figure 2.2. The diagram indicates that most of the mechanisms are limited in Category-I. Representative works include the Winsock2 Generic QoS API [96] of the Windows OS and the QoS enhanced socket [1], which provide QoS capabilities to socket abstraction. Crossing of X-axis is made by implementations like [6, 64], which provide packet classification by user ID and group ID, allowing control based on internal resources of the system. The Y-axis is crossed by the same implementations, because they offer packet filtering capabilities. However, no mechanism in this class covers the bottom right quadrant, Category-IV, which is to preserve resources from internal resource point of view, as exemplified by the assumption that a process should not exceed its granted resource usage. Because this property is important for system consistency, let us name it the *process-should-not-exceed* argument in the following discussion.

II.2.2 Class-B Services: Security Control

Almost every operating system nowadays is equipped with a software firewall to protect the system from network-borne threats. These mechanisms are representative of Class-B in the sense that they protect resources on the system from unwanted use.

Windows developers have been targeting their dominating client use, in the design of

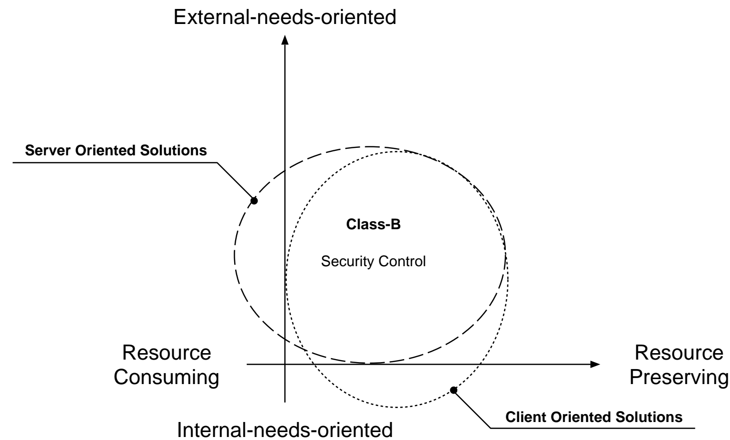


Figure 2.3: Class-B mechanisms

such mechanisms, whereas Unix gurus have been focusing on mostly low-end routers and servers with routing capability. Accordingly, these mechanisms are categorized roughly into two groups: client oriented mechanisms [35, 82, 84] and server oriented mechanisms [111, 73, 6, 114, 64].

As suggested in Figure 2.3, however, the security mechanisms are incomplete. The mechanisms on the Unix derivative operating systems are designed to protect the system from external traffic, and not to cooperate with internal resource management mechanisms to protect the system from internal threats. A simple illustration is the *process-should-not-exceed argument*. Class-B mechanisms cannot meet such a requirement, because of the lack of association with the OS resource management model. Note that controls based on process ID are insufficient, because a process may fork other processes and the group may exceed the resources granted to the original process.

In contrast, personal firewalls on the Windows OS have developed mechanisms which do have such associations, and allows security control based on the monitoring of process behavior. These mechanisms are, however, mostly made to cope with security threats, and do not have traffic control features, except for simple accept or deny control. Although their monitoring capability could have been used for adaptive control, they cannot be used for such purpose. These mechanisms also fail to solve the process-should-not-exceed problem.

Mechanisms specifically designed to protect CPU cycles [109, 120, 40, 27] also can be included in Class-B mechanisms. Unfortunately, they are mostly designed for CPU scheduling, and scheduling of network I/O is not their focus.

II.2.3 Class-C Services: Resource Management

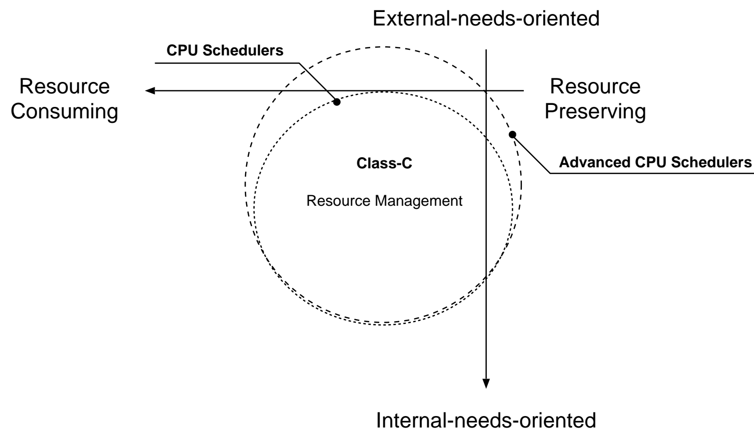


Figure 2.4: Class-C mechanisms

The third category is internal-needs-oriented, resource-consuming functionality, which is what CPU and I/O schedulers provide (Figure 2.4). Representative works include resource management mechanisms like [89, 26, 10], in the sense that they are designed to properly direct internal computing resources toward control goals, mostly for time-sensitive multimedia applications (*CPU Schedulers*, in the figure).

There are more advanced proposals, CPU resource management with bandwidth considerations [83, 124, 62, 59, 60]. These works originated in multimedia OS research like [89] in the 1990's, and provide guaranteed service, applying real-time OS technologies into the processing of network protocols (*Advanced CPU Schedulers*, in the figure).

Although they provide a flexible resource management scheme, they still have limitations, as suggested in the diagram: these mechanisms do not offer the capability to protect the system from outside threats. Another limitation is their resource management model. Some works have studied CPU resource consumption used for protocol processing [10, 72, 9, 26], but lack models for bandwidth and queuing control. Others have studied control of bandwidth

[57, 37, 65, 113], but lack a resource management model. A typical case is extension of the socket API, like Windows GQoS [96] and QoS enhanced socket [1]. Extension of the socket is useful for control of independent flow, but not a proper solution for control of *a set* of flows or for resource protection.

II.2.4 Class-D Services: Access Control

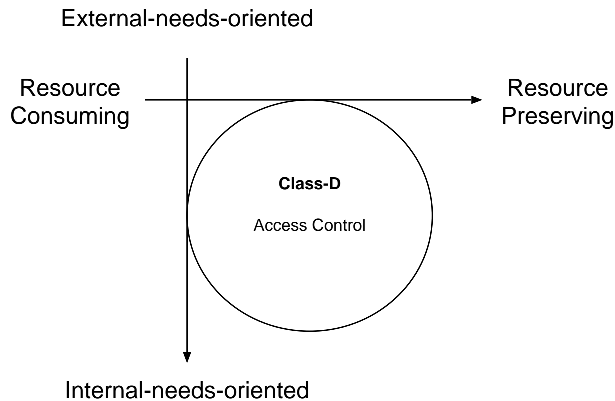


Figure 2.5: Class-D mechanisms

The last category of the classification is internal-needs-oriented, resource-preserving functionality, a functionality provided by access control mechanisms (Figure 2.5). An example is *tcpwrapper* [127], which controls connection establishment in the userland. Personal firewalls [35, 82, 84], reviewed in the Class-B discussion, also offer this functionality.

An obvious limitation of the mechanisms is that they can not be extended to provide greater controllability over traffic. Obviously, these mechanisms are designed solely for protective control, and thus, active control of network I/O is out of their focus.

II.2.5 Limitation of the approaches

As we have reviewed, there have been many mechanisms that are targeted at specific controls, but no mechanism covers all of the functional categories. This looks like a significant problem. An analogy is a car just with an accelerator. For proper functionality of the vehicle, we need brakes, as well as the accelerator. Likewise, there is a gap between the internal needs and

the external needs. Abstractions to meet external needs, such as flow abstractions, do not satisfy internal needs to control system resources, such as processes and threads. Meanwhile, abstraction of the system's resources do not satisfy the external needs.

However, a natural question arises here: "Isn't it possible to combine several mechanisms together, to overcome their functional limitations?" Actually, modularity of components is a reasonable approach to reduce system complexity, and we have realized a variety of applications by combining a limited set of independent mechanisms. Hence, it is natural to think that we may somehow combine the existing mechanisms to cover all of the functionalities.

The limitation of this approach is fourfold.

1. If we use mechanisms tightly associated with each system entity, we need to have independent mechanisms to address each of the possible conflicts and control targets in the system. For example, a process may want to control an independent flow, by limiting the bandwidth of a socket, whereas the user of the process may want the whole process to conform to a bandwidth limit that the user granted to the process. This is the process-should-not-exceed argument and we need a mechanism to address the problem. Meanwhile, a multi-threaded program may accommodate several connections per process, which calls for a mechanism to manage the concurrent connections of a process. It is costly to have dedicated mechanisms for each of these possible conflicts and control targets. We need *a general mechanism*.
2. Similarly, suppose that we have 1000 concurrent connections with intermittent traffic, or 1000 successive short connections. In these cases, control on each connection would have high overhead. It would be a natural approach to have a mechanism that properly aggregates the connections into a more coarse-grained abstraction, such as a socket group. Unless we have a general mechanism with flexible control granularity, we need to invent independent implementations for each control granularity. This again increases system cost and compromises application portability, which is what is happening in the network subsystem of modern operating systems. We need a *flexible control granularity*, which is hard to realize just by combining existing mechanisms.
3. There must be a protection mechanism so that an application can control only communications which are related to the application. This is a reasonable assumption, since

otherwise it would compromise the system consistency. Such a feature is, however, not realized by existing mechanisms. Accordingly, this control is never realized simply by combining them. We need a protection mechanism that properly *partitions resource consumption*.

4. In general, special purpose mechanisms are justified by their performance advantages, because their performance can be tuned for specific purposes. Combining such mechanisms may spoil the advantage, if there exists *an efficient general purpose mechanism*. For example, a user may want traffic control and packet filtering, both of which require packet classification. A general purpose mechanism may appropriately avoid the redundancy and may outperform the combination approach.

Consequently, the combination approach is unfavorable in many cases. Instead, there is a strong need for a mechanism which possesses i) generality, ii) flexible granularity, iii) partitioning of resource consumption, and iv) efficiency. To this end, we first need a general control model, which can simultaneously satisfy these goals.

II.3 STRUCTURAL ANALYSIS OF NETWORK I/O IMPLEMENTATION FRAMEWORK

To overcome the identified limitations in network I/O of modern operating systems, we need to modify the packet processing code in the operating systems. However, most packet processing mechanisms have been directly implemented on the operating systems. Because of the customary *hardcoding* of packet processing code in operating system kernels, these these services lost flexibility and portability.

This section reviews past efforts that have tried to unbind the tight coupling between operating systems and their packet processing code, in the aim of overcoming difficulties in the hardcoding. To present the approaches in this field, a given framework may fit too many of the categories in the classification system used in the last section, because they are designed to flexibly provide a variety of functionality. Accordingly, in this section, the related works are classified by different systems, not by the classification system presented in the last

section. First, they are classified by the location: in the kernel or userland. Subsequently, they are categorized in the order of abstraction level, starting from the restricted ones to highly flexible frameworks.

II.3.1 Processing location

We review mechanisms to realize various services for network I/O with respect to their location. In this area, there are two major approaches to take: implement the services in userland for flexibility or make the kernel extensible. Below, each approach is examined.

II.3.1.1 Userland protocol processing The first approach is called Application Level Framing (ALF) [15, 90, 44, 22], also known as userland protocol processing. This allows applications to execute their own packet processing code, customized for their needs, by providing raw access to network interface. Most of the work using this approach is motivated by performance gain, through specialization of protocol processing code.

To support such an approach, a variety of mechanisms have been proposed. The Virtual Interface Architecture (VIA) provides abstraction of the network interface for efficient raw device access [45]. U-Net is another approach for virtualized network interfaces [129]. There are also buffer management schemes [5, 32, 42, 133], most of which eliminate costly buffer copies between system boundaries to efficiently transfer data between the application, the kernel, and the device drivers. There are also approaches [49, 75] that transfer the entire service responsibility to userland, like a microkernel [110].

These approaches achieved far better performance for network I/O processing, as well as greater flexibility. However, as reviewed in the introduction chapter, an end-host accommodates various users and applications together, each of which has their own control goals and preference in the resource consumption. Accordingly, unless the kernel provides resource management and protection, system consistency would be easily compromised. If resource management can be safely transfered to users, as explored in the Exokernel [49], it is possible to choose to transfer the responsibility entirely to applications, in addition to the code execution. In a general purpose operating system, however, the kernel needs to hold

the control.

Application Level Framing is a best match for performance conscious systems for limited purpose in that it allows detailed specialization of packet processing. However, general purpose operating systems accommodate various users and their applications, which require appropriate resource management among them. Accordingly, although it unbinds the connection between operating systems and their packet processing code, it is not a desired form of network code implementation for general purpose operating systems.

II.3.1.2 Kernel extensibility The other approach for flexibility in packet processing, kernel extensibility, is a concept that allows flexible expansion of kernel functionality [118]. This approach is also beneficial for performance, because it can avoid data transfer between the kernel and userland by performing necessary processing inside the kernel.

Researchers have been studying such kernel extensibility for many aspects of kernel components, including communication processing. Stream [112] is an attempt in this direction, which allowed system users to insert processing modules into the data path between a device and a terminal connected to it. SPIN explored application specific protocol code which is dynamically loaded into OS kernels [53, 55, 54]. x-Kernel [66, 70] is an attempt that provides an explicit architecture for constructing network protocols in the kernel. Even on general purpose operating systems, there are frameworks for flexible protocol implementation, like Netgraph [46], which is in wide use even for production systems. It is also relatively common to support flexible extension of kernel functionality *via* “kernel modules”. For example, Dummynet [113] can be dynamically loaded to FreeBSD [58], using such an extension mechanism. It is also a common approach to support flexible extension of packet processing code, as a virtual device driver. In this approach, users can insert a module into the data processing path, as is used in the tunneling of packets and in some “personal firewall” systems [84, 82].

The drawbacks of these mechanisms are fourfold.

1. First of all, the control granularity of such a mechanism is too inflexible. Modules are inserted into the main data path, and the processing is applied to all the traffic that passes through the point of control, indiscriminately. A typical case is control per

network interface, which does not allow control of process network I/O or even more fine grained control. It is clear that such a mechanism cannot meet the requirement of flexible control granularity identified in the last section.

2. Even if the kernel module approach may provide mechanisms for flexibility in control granularity, such an approach creates tight association with underlying operating systems, which compromises portability of the code. It might be possible to provide a portable kernel API. However, without a mechanism for resource partitioning and access control, it would be limited to administrators.

3. As suggested just above, they allow raw access to kernel internals and lack resource management framework. The former necessitates a protection mechanism, not to jeopardize the entire system just by a misbehaving program. The latter requires appropriate partitioning of resource consumption. Otherwise, such a mechanism would be only for use by administrators.

4. Lastly, as an implementation framework, some of them are designed for overly specific purposes, such as protocol processing. For example, x-Kernel [66] provides a reasonable framework for protocol implementation, but it is not a good fit for flow control.

To sum up, existing frameworks have several shortcomings. First, they are deficient in that control granularity is fixed. Second, they form another OS boundary. Implementation framework must bring about code portability and OS independence. Third, some of the frameworks allow raw access to kernel internals. Appropriate hiding and abstraction of kernel structures are needed. Lastly, abstraction for specific purpose spoils flexibility and generality of the framework. These facts suggest the need for *appropriate virtualization scheme of network I/O code*.

II.3.2 Degree of virtualization

The observations in the last section suggested that excessive abstraction restricts flexibility (in the x-Kernel case) and too much flexibility spoils portability (in the case of ordinary kernel modules). Because both the extremes are harmful, a discussion of the appropriate degree of

virtualization is necessary here. To this end, this section reviews existing approaches in the order of virtualization degree, starting from limited ones through unrestricted ones.

II.3.2.1 Packet filters The first example of a limited environment for packet processing is "packet filters", also known as packet classifiers, filter micro-machines, and packet demultiplexers. They are used for packet demultiplexing and for selective monitoring of network traffic. Although they are designed specifically for efficient classification of packets, it is a form of in-kernel execution environment for user-supplied code.

This field was pioneered by an epoch making study, the Packet Filter [91]. This mechanism was designed to efficiently demultiplex received packets in the kernel, for efficient implementation of user-level network code. The Berkeley Packet Filter [88], the de-facto standard packet capturing mechanism, is a descendant of the mechanism. In addition to these studies, there have been a variety of proposals in this field [7, 47, 140, 67, 19, 13].

However, they have only limited use: classification of packets. To break this limitation, some researches proposed to extend the functionality of the micro-machines, for example, to allow efficient processing of traffic statistics and packet filtering inside the kernel [67, 101].

These proposals show that it is possible to allow execution of user-supplied code inside the kernel given proper protection mechanism. However, current approaches are still aimed solely at passive monitoring of traffic, and do not provide means to actively control data flow.

II.3.2.2 Structured approach in protocol processing Secondly, there exist a series of research proposals for formal and modularized approaches in protocol processing. Modularity is inherent in protocol processing, and thus, they mostly encapsulate each processing layer into an independent software component, abstracting the relationships among them.

For example, O'Malley presented a modeling method for protocols [103], Braun presented a protocol compiler for distributed applications [22], and Castelluccia investigated generation of optimized protocol stacks out of formal protocol specifications [28].

As shown, many proposals were made for many aspects of protocol processing in 1990s, and these approaches contributed to the design and implementation of protocol stack for the

following reasons. First, these approaches made packet processing code portable. Secondly, they explored abstractions needed for protocol processing, such as protocol graphs, which unbind protocol code and kernel internal structures. The downside is that they are designed for a very specific purpose, protocol processing, missing flexibility and generality. These approaches do not consider control of the traffic they generate, such as queuing control and packet filtering. Also, resource management is rarely achieved in these schemes. They are indiscriminately applied to all the traffic on the host, and thus control granularity is fixed.

Accordingly, these approaches have limited utility for actual network I/O implementation, and they are not in use on any general purpose operating system in existence, to the best of our knowledge. These approaches would be better applied to systems with specific purposes. For example, developers of embedded systems and experimental systems experience difficulties building a working protocol stack for their operating systems. Developers of such systems would benefit from the structured approaches, to keep development cost of their operating systems low.

II.3.2.3 In-kernel execution of packet handling code Packet filters provide execution of OS-independent packet processing code, but they have limited use. Accordingly, there is another set of approaches aimed at providing further flexibility in the processing. Indeed, several researchers have tried to execute user-provided packet processing code inside kernel, while providing the OS-independence.

A typical approach is user-provided interrupt handlers. ASHs [131], which stands for Application-Specific Handlers, proposed to execute event handlers to process packets without moving packet data between kernel and userland, for high performance messaging. U-Net/SLE (Safe Language Extensions) proposed to execute such a code on a hardware interrupt, by implementing a simple Java interpreter in a device driver [134, 136, 135]. A notable feature of the work is that they extended the Java specification to securely execute the user-supplied program in kernel. A descendant of U-Net/SLE [132] utilized an Ahead-of-Time (AOT) compiler to further boost the performance of Java bytecode execution.

Their contributions are twofold. First, they simultaneously achieved flexibility and performance gains, by providing another form of kernel extensibility for application specific

tuning. Second, the event handler model can be portable. On the other hand, they share the same limitation as most of the other approaches, in the lack of a resource management model. Further, because they work at a fixed location in packet processing path, they are indiscriminately applied to all the traffic.

This was a trailblazing field in in-kernel execution of packet processing software components supplied by users. However, they do not settle the problem of resource management and control granularity. Accordingly, we need to further extend the flexibility so as to implement a compete set of network I/O mechanisms that may also solve the problems.

II.3.2.4 Kernel extensibility revisited The observation suggests that we need further flexibility, and this brings us back to the kernel extensibility approach. However, as we have already seen, current kernel extensibility mechanisms have limitations. For example, raw access to kernel internals results in compromised code portability and system security. Loadable modules have architecture dependency problem. As Druschel pointed out [41], flexibility can be harmful in some cases.

To address these problems, we need to hide unnecessary kernel internals and to introduce a layer that virtualizes the physical instruction set architecture. Examples are Java and Flash, in the application area, which have achieved architecture neutral environments for code execution. Such a virtualization framework for kernel extensibility would break the binding between a service and the underlying operating system, and would favor flexible expansion of kernel functionality in a "OS-independent" manner. Further, the technology realizes easy deployment of advanced technologies onto systems in use, bridging the death valley which lies in between the research field and the market.

It is, however, not as simple as it might seem. First of all, we need an implementation framework which realizes resource management and flexible control granularity. Second, an appropriate level of abstraction must be determined. As illustrated just above, increased abstraction level results not only in good portability, but limited extensibility as well. Decreased abstraction level may allow more flexible implementation, at the cost of poor portability and compromised security. Third, to guarantee system safety and security, we need to add a layer, which may, in turn, sacrifice performance.

The hardcoding of packet processing code inside the kernel inflates the cost to add new functionality and to improve the implementations. For popularization of a new service, we are forced to manually port the mechanism onto major operating systems, which prohibitively increases the cost for technology deployment. To break these limitations, network I/O code must be virtualized in a manner that achieves a resource management framework, flexible control granularity, and OS-independence, while providing practical performance.

II.4 SUMMARY - "NETWORK I/O HAS BEEN POORLY VIRTUALIZED"

The network subsystem on end-nodes is a singular point. All of the communications on the host pass through this choke point, and there is no alternative to enforce appropriate control over traffic for end-host applications and users. Accordingly, packet processing at this locus has crucial importance to network systems, and thus, enormous attempts have been made to address a variety of problems. This chapter classified such approaches and clarified what they did achieve, and what they did not achieve.

To this end, a classification system based on service functionality was first presented. The system classified a variety of network I/O services into four major categories, and clarified that existing mechanisms with fixed control targets are not ideal as a control mechanism for network I/O on the end-host. First, they lack generality as a control mechanism. Second, they lack flexibility in control granularity. Third, they are used only by administrators, and partitioning of resource consumption is needed. Lastly, efficiency is needed. Accordingly, there must be a control model of network I/O that can meet these requirements.

Subsequently, proposed frameworks for network I/O implementation were reviewed. First, the chapter presented several proposals, classifying them by location: kernel or userland. The systematic investigation suggested that kernel approach is more appropriate than userland approach in respect to resource management. Second, this chapter demonstrated that existing approaches can be classified by degree of abstraction, and presented them in the order, starting from limited ones to more flexible approaches. The systematic review suggested that, although higher flexibility is desirable in general, it compromises portability

and security. On the other hand, increased abstraction level results in good portability, at the cost of limited extensibility. Requirements for an alternative are threefold. First, we need an implementation framework that possesses resource management model and flexible control granularity. Second, appropriate level of abstraction must be determined. Third, we need a virtualization layer that hides unnecessary kernel internals, guarantees system safety, and provides architecture neutral code execution.

To sum up, we need a new design principle that guides the design of network I/O of end-host operating systems. First, we need a new virtualization model that realizes consistent and complete resource management. Second, it calls for an appropriate mechanism that allows flexible implementation of network I/O code. The next chapter investigates the virtualization model.

III VIRTUALIZATION MODEL OF NETWORK I/O

The last chapter illustrated that there are two technical challenges in virtualization of network I/O, both in its abstraction and in its implementation. This chapter addresses the abstraction part, and presents a novel virtualization model, *hierarchical virtualization of the network interface*.

This chapter first discusses several possibilities for the abstraction. Then, the hierarchical virtualization model is presented. This model virtualizes a physical network interface to a hierarchical structure of virtual interfaces, and connects the virtualized interfaces into terminating OS entities, such as sockets and their owner processes.

We examine the virtualization model further, and illustrate that it realizes a variety of network I/O controls with flexible control granularity in a unified resource management framework.

The last section presents a qualitative evaluation that shows that the virtualization of network I/O possesses various advantages over existing attempts, and possesses desirable properties as an abstraction in this problem domain. The quantitative evaluation is described in the next chapter along with prototype implementation of the model.

III.1 ABSTRACTION FOR NETWORK I/O VIRTUALIZATION

To virtualize network I/O on end-host operating system, the simplest approach is to abstract *end-to-end communication* and provide control over the abstraction. For example, the socket abstraction is a way of abstracting end-to-end communication, and one might extend it to support more active control over it. However, because we may want to control

a set of flows, not just the flow associated with the socket, it is also possible to provide an aggregated abstraction, raising the abstraction level. Accordingly, in the virtualization of end-to-end communication, there are antithetical approaches, namely, low-level abstraction and high-level abstraction. However, we contend that both of the approaches have fundamental shortcomings as abstractions of end-to-end communication, as summarized below.

III.1.1 Abstraction of end-to-end communication

III.1.1.1 Low-level Abstraction One way to control network traffic is to provide control services that work directly over flows. For example, we may extend the socket abstraction so that users can set bandwidth limitation. Admittedly, the low-level control over each connection is intuitive, and it would be useful for some applications. However, as enumerated below, there are fundamental problems with this scheme.

1. Such an interface can *easily contradict the resource management semantics of the operating system*. For example, a network-intensive low-priority process with a large bandwidth limit can easily starve higher priority processes that require network I/O. This compromises the resource management semantics of the OS. Clearly, *resource protection* is needed for the system to be consistent.
2. *The low-level abstraction is not scalable*. Suppose that we have 1000 concurrent connections with intermittent traffic. It is quite inefficient to designate fixed network resources to each of them. In this case, proper traffic aggregation can greatly improve the resource utilization, as well as the quality of each session. Likewise, if we have 1000 successive short connections, we may want to assign an aggregate flow specification for all of them. Clearly, *aggregation* is needed.
3. The low-level abstraction scheme *presupposes knowledge of independent connections*, which is not always possible. For instance, if we are to control output of a web browser, we need to know how many connections the browser makes and when. To address the problem, we need another system service to properly locate the connections they make and monitor the usage. This would probably increase design complexity of the system, and cause extra overhead with latency.

4. In the area of network resource management, *the service model is still an open problem.* For example, we might have service models such as IntServ and DiffServ, but there are also models for authentication and accounting. Consequently, the most probable scenario is that each administrative domain chooses its own models and services to offer, based upon its own needs and resource constraints. Accordingly, it would harm portability of the applications, if particular semantics is added at the OS layer.

As shown, although the low-level abstraction of flows provides fine-grain control, it is not always the most desirable solution. Consequently, we need a higher level of abstraction.

III.1.1.2 High-level Abstraction An example of higher level abstractions for end-to-end communication is a device file under `/dev/network/`, which might support resource management and flow aggregation. Although this approach also sounds reasonable, it is not as simple as it may seem at a cursory examination. Fundamental problems of this scheme are threefold.

1. First is the *lack of controllability* of the network. Although the network interface is one of the major I/O devices of a modern computer, the network I/O differs from typical I/O access, such as those for hard drives and serial ports, in that most of the network I/O involve remote resources. Accordingly, controls on the abstraction could be inaccurate and inappropriate in many cases. For example, the bandwidth guarantee might require output rate regulation at the host and a path bandwidth reservation, which is the network's responsibility. This fact does not make a single and consistent abstraction by the end-host OS impossible, but clearly makes it harder to achieve.
2. To provide such a high-level abstraction, we *need a standardized abstraction and interface* for programs. However, there have been a variety of network protocols and it is also hard to predict their future forms. For example, unicast, multicast, and anycast differ greatly, and thus, it is hard to converge them into a single abstraction. Further, there are various irregular protocols, such as the Real Time Streaming Protocol (RTSP) [115], which has a connection for data streaming and keeps another for health check of the stream. It is hard to statically define a single standardized abstraction and system service which

satisfies all the various needs.

3. A service that involves abstraction of end-to-end communication requires close cooperation of kernel and user-level applications. Let us take the example of path bandwidth reservation. We may employ an RSVP [108] module at the user level for end-to-end signaling, but in some cases, we also need a kernel module to regulate the host's network I/O. This poses another question of how to separate the functionalities between kernel and user-space.

As briefly illustrated, it is quite challenging to have a system service for *high-level abstraction of end-to-end communication*. It was also suggested earlier that the low-level abstraction of end-to-end communication, which might represent a single flow, is not a reasonable option to take. The conclusion we reach from the two perspectives presented is that *abstraction of end-to-end communication* is not appropriately supported by operating system. This suggests that an *abstraction model of network I/O that works just inside the end-host* could be more reasonable one.

III.1.2 Virtualization inside end-host system

There have been many abstractions of network I/O on end-host operating systems that work solely at the end-host systems, as summarized below. Indeed, we have been *virtualizing network interfaces* in many ways that serve as abstractions of network I/O.

III.1.2.1 Horizontal virtualization of the network interface A simple illustration is assignment of multiple network addresses on a network interface, also known as *aliasing*. This way, we can virtualize a network interface into several entities, and can independently control them. In a more elaborate manner, there have been implementations that provided virtualized network interfaces to performance-conscious applications for efficient raw I/O access [129, 45]. VMware [128] can emulate virtualized hosts with virtualized network interfaces, connecting them even in a virtualized network segment [122]. Even the "socket" abstraction can be seen as virtualization of network interface.

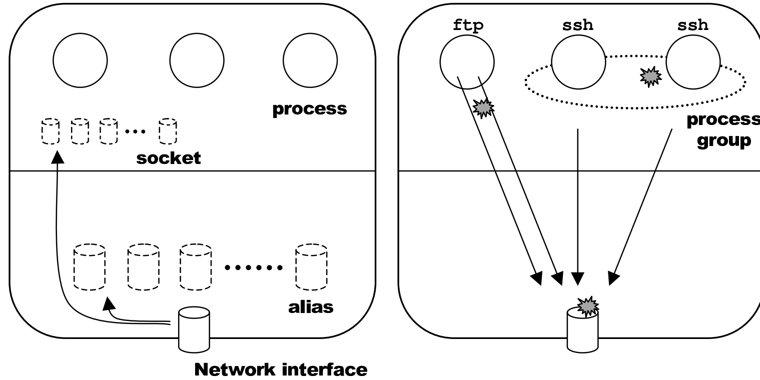


Figure 3.1: Horizontal virtualization and potential conflicts

These virtualizations of network interfaces are mechanisms that are completely inside end-hosts. Although they differ in their goals and abstractions, all of the approaches share one important property: they multiplex a network interface *horizontally*, into multiple entities (Figure 3.1-left).

However, the simple *horizontal virtualization* has shortcomings in its resource management. For example, a FTP process for bulk data transfer may compete with SSH processes for interactive traffic, while the several connections for the FTP process may also be at odds with each other. Because there are a variety of entities that can potentially terminate the connections (e.g., sockets, their owner processes, or process groups on a host), conflicts for the shared network interface may happen at various granularity (Figure 3.1-right).

Accordingly, in the resource management point of view, the horizontal virtualization is incomplete. As indicated in Chapter II, we need *proper partitioning of resource usage* among various users and applications, avoiding interaction of independent entities.

III.1.2.2 Hierarchical virtualization of the network interface To avoid interaction between running processes, we may virtualize the network interface and attach it to the network I/O of each process [99], as shown in Figure 3.2-left. The virtualized interfaces partition process activities, and provide protection among processes.

However, there are several problems in this configuration. First, communications within a process are still in competition. Second, even cooperating processes are managing their own

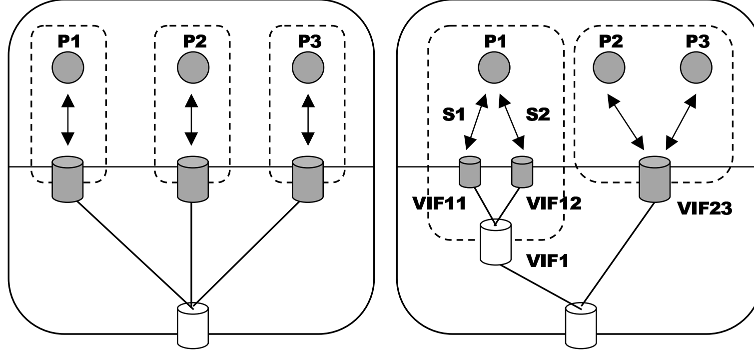


Figure 3.2: Hierarchical virtual network interface model

resources independently, compromising efficiency and resource utilization. These problems happen because granularity of the partitioning is fixed to process level, and this becomes just another form of the horizontal virtualization.

Accordingly, for flexibility of the control granularity, we extended the model as follows. First, the model is extended to allow recursive creation of the virtualized network interfaces. Second, the model is extended to allow attachment of the virtualized interfaces (VIF) onto terminating entities, such as sockets and process groups (Figure 3.2-right). Process 1 (P1), first connected to VIF1, created VIF11 and VIF12 *on top of* VIF1, to control two sockets (S1 and S2). This model realizes independent control over socket 1 and 2 from Process 1, within the limit of network resources assigned to VIF1. This model also favors sharing of a virtualized interface among cooperating processes. Suppose that Process 2 (P2) and Process 3 (P3) are web servers. We can control all the web traffic just by having the common interface VIF23 and by performing necessary control over it, instead of maintaining independent interfaces.

This structured creation of virtualized interfaces is called *hierarchical virtualization* of the network interface. This simple model accomplishes partitioning of the physical resource with flexible granularity, not limited to just interfaces or processes as terminating entities. This local mechanism also works independent of application protocols and of the communication topology (such as unicast, and multicast). This suggests that the hierarchical virtualization model possesses desirable properties as an abstraction of network I/O for end-host systems.

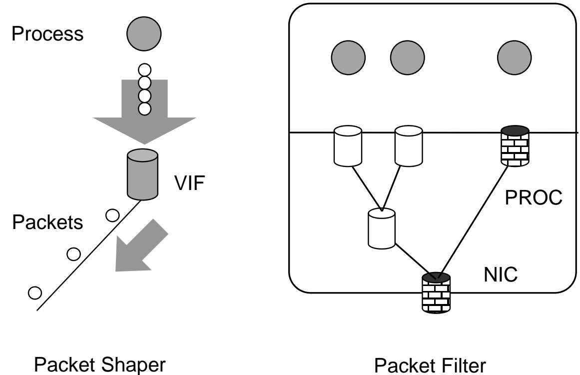


Figure 3.3: Various virtual network interfaces

III.2 HIERARCHICAL VIRTUALIZATION OF NETWORK INTERFACES

To be a practical model for network I/O control, the model still needs to consider several issues: control generality, resource protection, and access control. Each of these topics are covered below.

III.2.1 Generality of Control

First, we introduce a simple principle in the control model; this model allows each VIF to do any control inside the VIF, except for modification of the packets. The simple rule brings about the vast diversity in the control it can realize.

For example, to control bandwidth, we can use a packet scheduler that does bandwidth control (Figure 3.3-left, where a cylinder denotes a VIF and small circles are packets in the diagrams). If we connect the VIF to a process, we can realize per-process control; if we connect to a socket, we realize per-flow control. Another example is a packet filtering; we can simply run a packet filter on a VIF for firewall service (Figure 3.3-right). In contrast to traditional software firewalls, this model supports security control with flexible granularity, such as per-interface (see NIC) and per-process (see PROC) [101]. It is also possible to have a VIF which blocks all the communication; a blocking VIF. The bandwidth control and the filtering service are essentially equivalent, in that packet filtering is to set the bandwidth of

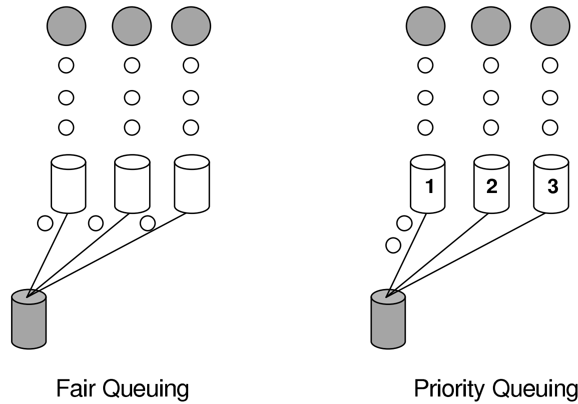


Figure 3.4: Various virtual network interfaces

the traffic to zero.

It is also possible to perform more elaborated queuing control, such as work-conserving packet scheduling, which makes efficient use of excess resource. A child interface of a Fair Queuing VIF can utilize the entire bandwidth of the parent VIF if there are no other competing interfaces (Figure 3.4-left). They do fair-sharing of the bandwidth automatically, if a conflict happens. A Priority Queuing VIF is a virtual interface which prioritizes child interfaces (Figure 3.4-right). In file-sharing software, for example, a user can dynamically prioritize his file transfers over others, with this type of VIF.

Note that, although simple bandwidth limiting is possible by a library, like [50], at userland, queuing controls like WFQ and PQ are performed only at the operating system level, where actual contention occurs. Also, dynamic control of on-going traffic by applications and users is hardly possible with existing mechanisms for network control on end-host operating systems, such as [113, 31].

III.2.2 Rules for Resource Protection

Because this control mechanism is powerful as was just illustrated, we need to somehow prohibit exploitation of the mechanism by users and applications. Otherwise, a process may declare the need for a small network use, and end up with much more than its share. This may happen, if a process with, say, 1Mbps limit reconnects a socket to a VIF with 100Mbps

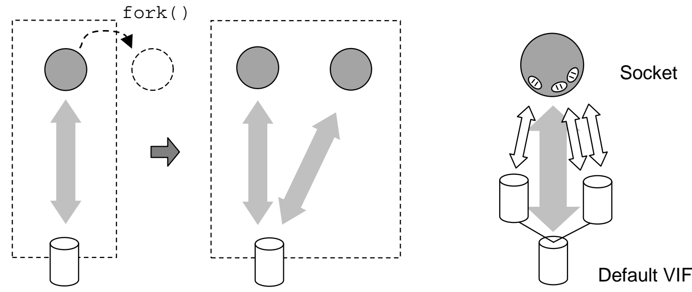


Figure 3.5: Resource protection and Access control model

limit. This would be clearly problematic and should be avoided.

To assure resource protection among VIFs in the VIF system, we need to introduce simple rules.

1. The hierarchical virtual network interface model introduces *ownership* of VIFs, prohibiting control of unrelated VIFs. Users can only modify parameters of the VIFs they own, whereas system administrators have privilege to modify all parameters.
2. Child processes inherit the VIF originally attached to the parent process (Figure 3.5-left). This rule prohibits exploitation of resources by indiscriminate forking. At the same time, this realizes simple access control. If the parent has access to a VIF, the children are allowed to use that interface, by the inheritance mechanism. Conversely, if a process does not have access to an interface, children will have no access to that interface, because they will inherit nothing.
3. Sockets can connect only to the VIF that the owner process is connected to (default VIF) or to its descendant VIFs. This rule guarantees that all the communication of a process is limited by the default VIF (Figure 3.5-right).
4. Users can only remove leaf VIFs that do no have any child VIF. This rule prohibits removal of trunk VIFs. This is an important property, otherwise we may leave disconnected VIFs in the system.

Following these rules, we can assure the consistency of the system and make resource units conform to the resource limit granted to them originally. If any violating operation is performed, the system may return an ”access denied” error. These protection rules allow

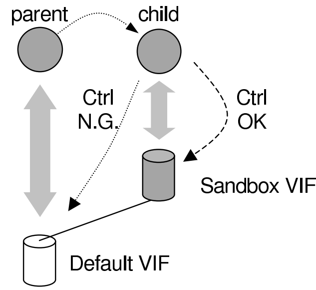


Figure 3.6: Sandboxing a virtual network interface

control of network I/O even to network-borne applications, such as Applets and Flash, by creating a sandbox (Figure 3.6). The sandbox VIF prohibits any control that violates the authority granted to them.

III.2.3 Access control and system interface

Another challenge is how to express the hierarchical nature of the VIF tree, in order to realize a proper Application Programming Interface (API). How is it possible to provide appropriate access right to users for VIF access? It is an indispensable part of the system, and at the same time, we need to avoid any increased complexity of the system or conflict with existing mechanisms.

As an answer to these problems, *file system abstraction* can be a reasonable approach. In file system abstraction, a VIF is represented by a directory, which contains parameter files. The hierarchical structure of the virtual interfaces is intuitively expressed in the structuring of the VIF directories. In this setting, ownership and protection of the VIFs can be naturally realized with the access control semantics of underlying file system.

As suggested, the file system abstraction is intuitive. Additional advantages include the following. First, the abstraction enables various types of network control in a single consistent interface. For example, the file system may provide a writable "type of VIF" parameter file, and users may simply specify a type of the VIF being controlled. Second, the API supports any control targets, as long as they are representable in the file system abstraction. For

example, by abstracting processes and sockets, we can control these OS abstractions in a unified manner. It is also possible to support control of threads if they are appropriately represented in the file system. Third, the API is accessible to any programming environment that supports a file access API, such as shell scripts, even without system call interface.

Consequently, the file system abstraction is considered as a promising system interface for the hierarchical virtualization scheme. In the meantime, it is still unclear how to specify association of a VIF and terminating OS entities, such as sockets and processes. Such issues related to the system interface were addressed when a prototype system of the hierarchical virtualization model was implemented, described in the next chapter.

III.3 QUALITATIVE EVALUATION AS A VIRTUALIZATION MODEL

After having presented the virtualization model, we evaluate it qualitatively. For this purpose, a case study of the virtualization model is first presented, in which both administrators and users are represented as well as interactive and bulk data transfers. This scenario is illustrated in Figure 3.7.

The system allocates independent VIFs to the login-shell of each user, thereby partitioning their resource usage. Resource consumption of User A, executing `ftp`, is bound by the VIF connected to the login shell. Another user, User B, is also executing a `ftp`, but through a command, `ftp ftp.freebsd.org @128Kbps`. An extended shell automatically creates a dedicated VIF with 128Kbps limit on top of the shell VIF, and limits the bandwidth usage of the network application. Although the `ftp` processes are controlled from outside of the processes, resource conscious applications may create private VIFs to control its own traffic with a finer granularity, on per-flow or on per-thread basis.

As illustrated, hierarchical virtualization of network interfaces accomplishes a variety of properties desirable for virtualization of network I/O on end-host operating systems. These properties, described below, are easily derived from the scenario above.

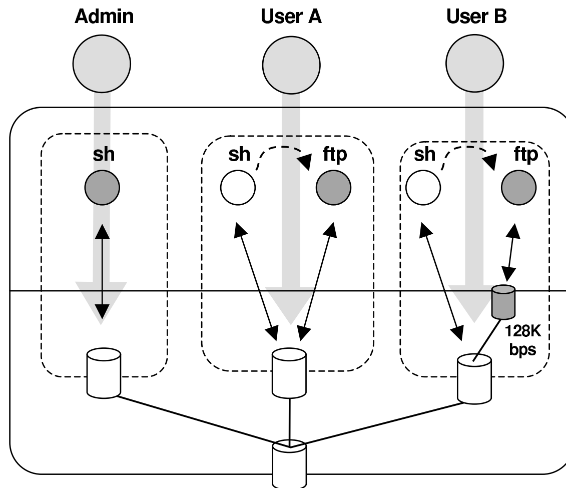


Figure 3.7: Sample scenario

1. It allows flexible network control by user commands and network applications, unlike existing systems designed solely for administrators. Note that all the existing network control mechanisms are protected by privileged system calls to prohibit indiscriminate access from users, severely limiting the potential of these end-host systems.
2. The hierarchical structuring of interfaces provides a consistent framework for resource partitioning and its management. Users and their applications are allowed to perform any control within the limits of the allocated resource.
3. The attachment mechanism provides a simple solution for flexibility in its control granularity, such as a process group, a process, or sockets.
4. The hierarchical virtualization of network interfaces provides great freedom in the control of network I/O. Because it is a virtualization of a network interface, it can perform any control of network I/O realized thus far on a network interface, such as bandwidth control, fair queuing, priority queuing, and security control.
5. This mechanism allows *retrofitting* control, in that the mechanism allows users to control existing programs without modification.

It is noteworthy that all of these advantages are attained just by a simple idea, hierarchical virtualization of network interface.

III.4 SUMMARY - "HIERARCHICAL VIRTUALIZATION IS THE ONE"

This chapter investigated virtualization model of network I/O in an end-host operating system. First, the chapter studied several possible approaches in search for appropriate virtualization of network I/O, showing that abstraction of end-to-end communication is inappropriate and suggesting a need for a model that works just inside the end-host. The existing mechanisms on end-host operating systems have a certain property in common: they multiplex a network interface into multiple similar entities, which we call *horizontal virtualization of network interface*. However, this approach is considered to have shortcomings in the resource management point of view. To overcome this limitation,the *hierarchical virtualization of network interface* is presented, which virtualizes a physical network interface to a hierarchical structure of virtual interfaces and connects the virtualized interfaces into terminating entities, such as processes and sockets.

Then, the proposed scheme was investigated in further detail, to be a practical model for network I/O control. First, the generality of the model is assessed, and it is considered to possess appropriate generality as a network I/O model. Second, issues related to resource protection are investigated, and simple operation rules are introduced for system consistency. Lastly, API issue is addressed and *file system abstraction* of the virtualized interfaces is proposed.

These discussions are followed by a qualitative evaluation, which showed that hierarchical virtualization of network interface possesses desirable properties for virtualization of network I/O on end-host operating systems, namely, control freedom, partitioning and protection of resource consumption, flexible control granularity, generality as a control model, and the retrofitting property.

Even though the conceptual idea is clean and clear, the implementation of such an I/O model on actual operating systems is a challenge, including performance concerns. Accordingly, here arises a question: is it possible to implement such a model while keeping the performance practical? The question necessitates empirical proof of the concept, which will be the focus of next chapter.

IV PROTOTYPE IMPLEMENTATION OF THE MODEL

The last chapter investigated virtualization models for network I/O and presented a scheme, hierarchical virtualization of network interfaces. Contrary to the simplicity of the model, the implementation of the concept is quite challenging.

First, it requires a flexible hierarchical packet scheduler. Second, it needs to provide generality and flexibility in the controls it realizes. Third, in addition to these properties, the model must be implemented in an efficient manner.

To empirically study these issues, this section first presents an actual implementation of the model. Then, sample applications are presented, for demonstration of the prototype implementation. Additionally, for demonstration of the extensibility of the prototype, new functionality is added to the base implementation. Lastly, these implementations are evaluated quantitatively and qualitatively.

IV.1 PROTOTYPE IMPLEMENTATION

In this section, an overview of the prototype implementation on FreeBSD 4 is presented.

IV.1.1 Implementation overview

The implementation overview is given in Figure 4.1. The figure indicates that the VIF system resides in between the network layer and the interface layer of the operating system kernel. Each of the functional components is briefly described below.

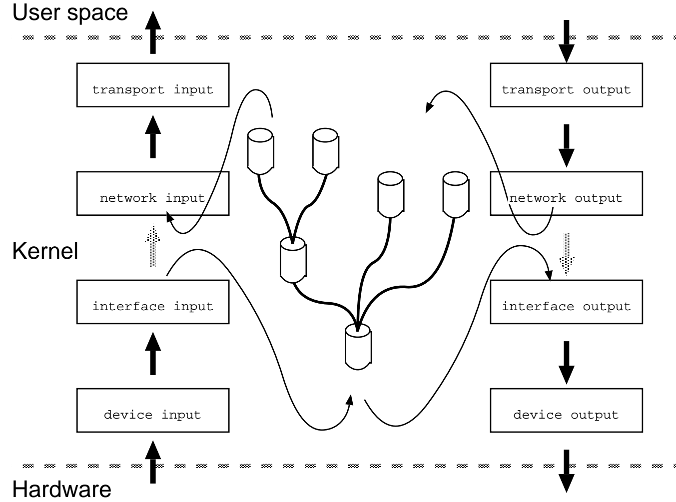


Figure 4.1: Implementation Overview

IV.1.1.1 Packet Hook The four thin arrows in the figure denote packet entry and exit points of the hierarchical virtual network interface tree. The packet hook mechanism resides between the network layer and the network interface layer. For example, if the Internet Protocol (IP) is used for network layer, we capture incoming packets before they reach the IP input function and capture outgoing packets after they are processed by the IP output function.

For each outgoing packet, after processing of network protocol (typically, by `ip_output()`), the kernel calls an interface output routine, that is, a function that will forward the packet to appropriate interface. In our implementation, the system redirects packets into the VIF subsystem, that is, the function pointer for `if_output()` is replaced by `vif_input()`.

The VIF subsystem consists of VIF data structures and two global schedulers, one for incoming traffic and the other for outgoing. A VIF data structure, `struct vifnet`, includes queue data structures, statistics variables, and scheduling parameters. Packets are inserted and *travel through* the VIF data structures connected to each other in a tree shape, originating the physical interface. After proper scheduling in the VIF tree, packets are again directed, by the `vif_output()` function, to the original outgoing function (`if_output()`).

Likewise, input packets are redirected from their normal path using function pointer

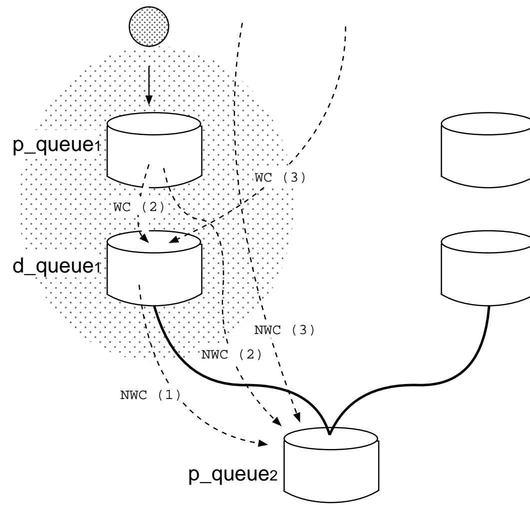
stealing. In FreeBSD, when a packet arrives at a physical interface, the kernel initiates a software timer for further processing by `ipintr()`. We overwrite this with `vif_intr()`, to divert the input packet. This way, input packets are redirected into the VIF tree and processed for further scheduling. After they reach the appropriate leaf VIF, the packets are returned to the regular upward processing stream.

IV.1.1.2 Packet Scheduling The main reason for redirecting packets into the VIF tree structure is to control the data traffic, typically by delaying or reordering the packets, that is, packet scheduling. Below we describe our scheduling framework.

The goal of the scheduler is to realize a single hierarchical scheduling framework that is able to accommodate both Work-Conserving (WC) scheduling and Non-Work-Conserving (NWC) scheduling modes. A Work-Conserving scheduler is one that processes packets whenever they are available in the queues, thereby contributing to better link utilization. A Non-Work-Conserving scheduler may defer the processing for better control of bursty traffic and predictability in its output.

The two different scheduling paradigms are unified in a single framework by two mechanisms. First, the system employs different drainage strategies for work-conserving VIFs and non-work-conserving VIFs. Second, a VIF is comprised of two queues: a proximal and a distal queue. Figure 4.2 shows a pictorial and pseudo-code description of the hierarchical, multi-discipline scheduling framework, in which packets *travel through* the VIF tree. Figure 4.2(a) is a VIF structure, where two VIFs have a common parent VIF (at bottom). As shown, a VIF is comprised of two queues, a proximal queue and a distal queue, labeled as `p_queue` and `d_queue`. Arrows in the figure show enqueue and drainage (dequeue) relationship among VIFs, which are labeled WC and NWC. The numbers in parenthesis next to the labels correspond to packet dequeuing operation of the scheduler (Lines 11-14).

Now we show how the system works, following the code. Suppose that the system enqueues a packet (depicted by a circle at the top of the figure) into the upper left VIF (in a shadowed oval). The system enqueues the packet into `p_queue` of the VIF (Line 3). Then, a global scheduler function, `vif_sched()`, is called (Line 4) to regulate the rate at which VIFs will be serviced; `vif_sched()` starts a software timer that calls the `vif_dequeue()` function



(a) Drainage strategies

```
1  vif_enqueue(vif, pkt)
2  {
3      insert_tail(vif->p_queue, pkt);
4      vif_sched(vif);
5  }
6
7  vif_dequeue(vif)
8  {
9      /* pick a packet */
10     do {
11         if ((pkt = vif->d_queue) &&
12              pkt->next_vif->type != WC) break; // (1)
13         if ((pkt = vif->p_queue)) break;       // (2)
14         if ((pkt = prev_vif->d_queue)) break;  // (3)
15         return;
16     } while (0);
17
18     /* enqueue to next */
19     next = pkt->next_vif;
20     switch (next->type) {
21     NWC:
22         if (next->root_flg)
23             vif_output(pkt);
24         else
25             vif_enqueue(next, pkt);
26         break;
27     WC:
28         insert_tail(vif->d_queue, pkt);
29         vif_sched(next);
30     }
31 }
```

(b) Drainage algorithm

Figure 4.2: The hierarchical packet scheduler

for each VIF at the appropriate time for packet dequeuing. The scheduling time of the dequeue operation is calculated from the packet length and bandwidth of the particular virtual interface. When the software timer expires, the system initiates the drainage operation, by calling the VIF dequeuing function.

In the dequeue function (Line 7) invoked by the software timer, packets are handled differently based on the scheduler type of each VIF (NWC or WC). Below, we illustrate the algorithm through a couple of scenarios.

A NWC VIF is chosen to generate constant bit rate flow, and for this purpose, packets are simply enqueued into the next `p_queue` at some configured rate. For example, let a new packet arrive at a NWC VIF (shadowed oval in Figure 4.2(a)), when all VIFs are empty. First, the system enqueues the packet into $\texttt{p_queue}_1$ (Line 3), and schedules the VIF for dequeuing (Line 4). Then, at the appropriate time, the scheduler calls `vif_dequeue()` at Line 7, which will find $\texttt{d_queue}_1$ empty (Line 11). Therefore, it falls through the `if` statement, and finds the enqueued packet in the `vif->p_queue` (Line 13). The algorithm then goes to the `switch` statement at Line 20, and sends the packet to output (Line 23) if the VIF is a root, or enqueues the packet into next VIF, $\texttt{p_queue}_2$, at Line 25. In other cases, the `do` statement at Line 10 may find a packet in a different queue (there are three possibilities, as shown), but a NWC VIF always sends it to `p_queue` of the next VIF.

A WC VIF is used for work-conserving scheduling, such as fair queuing and priority queuing, where packets are sent whenever the link is available. For this purpose, the VIF utilizes a special purpose wait queue, `d_queue`. The VIF puts packets ready to be serviced in the queue and wakes up the next VIF for their drainage. To demonstrate this, let a new packet arrive at a WC VIF, when all VIFs are empty. First, the system enqueues the packet into $\texttt{p_queue}_1$ (Line 3) and schedules the VIF for dequeuing (Line 4). Then, the scheduler calls `vif_dequeue()` at Line 7 and in the `do` statement (Line 10), we find the packet at Line 13. The algorithm, then, goes to the `switch` statement at Line 20 for the next operation, and inserts the packet into its own `d_queue` at Line 28, scheduling the next VIF for future drainage (Line 29).

In other cases, the `do` statement at Line 10 may find a packet in a different queue (there

are two possibilities, as shown), but it always sends the packet to its own `d_queue`, leaving the dequeuing operation (`vif_dequeue()`) to the next VIF. The dequeue function is called when the next VIF finishes its current operation, thereby realizing the work-conserving behavior of the VIFs.

IV.1.1.3 Packet Marking Since the overhead of packet classification can be overwhelming, we use a *packet marking* mechanism. This mechanism is used also for the *routing* of the packets in the VIF tree, as follows.

We modified the OS internal data structure for processes and sockets to include pointers to the attached VIF data structure[1]. We copy this pointer onto each outgoing packet, when the kernel allocates buffers for the data. Then, when the packet enters the VIF system, the kernel traverses the VIF tree from the associated VIF down to the root VIF creating a list of VIFs on the path. The list is attached to each packet, which is, in turn, enqueued into the corresponding VIF. The dequeue function, called by the scheduler, simply checks the list for next VIF without any detailed inspection of the packet for classification and enqueues into the next VIF by calling `vif_enqueue()`.

Incoming packets are handled differently, since they must be put into the root VIF first and traverse the VIF tree to reach a leaf VIF attached to the destination socket. Clearly, exploring the entire VIF tree would yield a severe performance penalty. Thus, we utilize *early demultiplexing*. When the packets emerge from the interface input routine, we first do a lookup in the Protocol Control Block hash table (in BSD terminology) and find the destination socket. Second, we find the VIF associated with the particular destination socket by using the pointer for its attached VIF. Third, we get the VIF path from the socket down to the root VIF. By reversing the order, we get the VIF list on the path from the interface upward to the target socket.

[1]Since we need to handle multiple physical interfaces, we actually attached a pointer to a list of VIFs associated with the sender, which is copied onto each packet. This list is used later to choose the VIF to enqueue when the packet enters into the VIF system, based on the routing information provided by the network layer.

IV.1.1.4 Tuning for better performance So far, we have presented a simplified overview of the implementation, for algorithmic clarity. However, since the VIF system lies in the packet processing path, it incurs extra overhead for system performance. We state several strategies we have used to improve performance, and suggest some points that can be exploited.

- We can cache the list of the VIFs on the path for each VIF to save the CPU cycles for the costly path list creation.
- In many cases, we can process consecutive packets together, to avoid the number of costly and inaccurate software timer events.
- We can demultiplex incoming packets at the VIF entry stage of the packet processing, and can save the packet classification data (a pointer to Protocol Control Block) for future use to avoid a redundant lookup operation later in the original location.
- We can greatly simplify the packet scheduling for incoming packets, since the upward path of the VIF tree has just a single data source (the root VIF), as opposed to the outgoing flows which have multiple sources. Note that, if there are multiple sources, we need an extra operation to choose a packet to process next each time.
- We can avoid the per-packet memory management cost, by preallocating necessary data space for the VIF system when each packet is created.

In our experience, a faithful implementation of the presented algorithm has caused a performance penalty of as much as one fourth of the peak throughput, or even worse. Meanwhile, as we demonstrate in the evaluation section, these optimizations contribute to make it approximately 10% in the worst case, and generally better than that, making it a practical option for most systems.

IV.1.2 System Interface

In the prototype, the system interface is implemented utilizing a *file system abstraction*. For this purpose, a file system, the Netnice file system (nnfs), was developed, reusing the file system framework of the process file system (procfs), a file system representation of process entries on a host [76, 51].

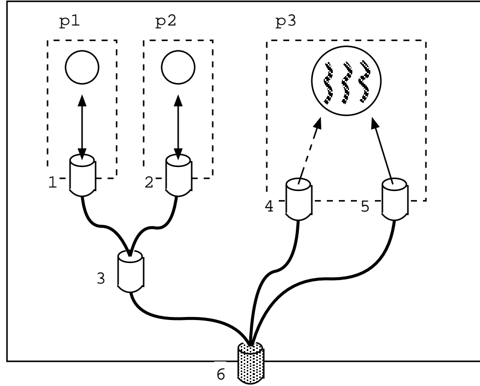


Figure 4.3: Sample configuration of a hierarchical virtual network interface structure

NNFS is typically mounted on `/proc`, and a VIF is represented by a directory, which contains parameter files. The hierarchical structure of the virtual interfaces is intuitively expressed in structuring of the VIF directories.

This section presents the file system API. For illustrative purpose, a sample scenario of a VIF-based network control is first presented. Then, file system representation of the control scenario is shown. Lastly, the operation model is presented, using common file management mechanisms.

The strength of the design will become clearer as the section progresses. Note that this is just a low-level API and we offer higher-level commands for daily use, which wrap the low-level system service, as demonstrated shortly.

IV.1.2.1 Sample Scenario Let us consider a VIF-based control for a typical client machine. In the scenario depicted in Figure 4.3, there are two daemon processes (p1 and p2) for network file system access. Suppose that they are mounting a file server on the local network, and have bursty traffic pattern (e.g.,file transfers). On the right is a web browser process (p3), accommodating several threads, one of which is reserved for data streaming. Note that bursty traffic by the network file system occasionally saturates the real network interface, and starves the web browser process flows. (In fact, the scenario was originally motivated by a similar problem we had in our department network [99].)

Our goal is to give proper protection to the processes, and provide better quality of

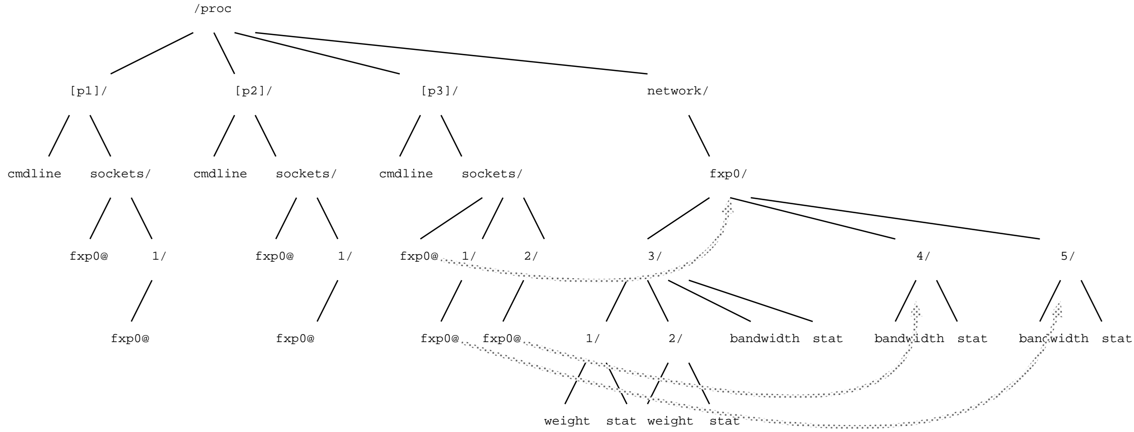


Figure 4.4: Directory structure for the sample VIF configuration in Figure 4.3.

service to system users. To this end, we show how to configure the VIFs below.

First, we reserve a certain amount of bandwidth for the streaming web application. This is achieved by limiting the file system traffic, and thus, we configure the flow specification of VIF 3 so that the bursty traffic is bound by some predefined bandwidth (e.g., 4Mbps). Next, we give fair shares to the file system processes for better utilization of the limited bandwidth; for instance, 50% of VIF 3 to each VIF 1 and VIF 2.

VIF 4 and VIF 5 are for web browsing. Let us suppose that VIF 4 is for the streaming, and VIF 5 for TCP connections. Again, to assure the streaming quality, it would be better to limit the bandwidth usage of VIF 5, leaving room for VIF 4 to receive the time-sensitive data.

IV.1.2.2 File System Abstraction Next we show how to carry out the configuration of the system, utilizing NNFS.

To illustrate the abstraction strategy, a sample directory representation for the VIF configuration shown in Figure 4.3 is presented in Figure 4.4. A directory under `/proc/network/` represents a VIF. The root VIF is named `fxp0`, which corresponds to the name of a real network interface on the machine, marked as interface 6 in Figure 4.3. The root VIF has three VIFs under it: VIF 3, VIF 4, and VIF 5. Files under a VIF (e.g., bandwidth and stat) contain configuration parameters for the virtual interface (for simplicity, only some of them

are shown in the figure).

Directories under `/proc/[pid]/` are process entries, as usual. We added a `sockets` directory under each process entry. As shown, process 1 (p1) and process 2 (p2) have one socket each, and process 3 (p3) has two sockets. Directory names (such as `1` and `2`) for the sockets correspond to the file descriptors they are assigned to.

Under each individual socket directory, there is a link named `fxp0`, indicated by the @ mark in Unix convention. The links denote attachment of the VIFs to the particular resources. For example, we can see that p3 has its network I/O capped by a VIF represented as `/proc/network/fxp0/`, and the two sockets of p3 are connected to VIF 4 and VIF 5.

The association of a VIF and an OS-supported entity (e.g., a process or a socket) is represented by a link with the same name as the real interface (in this case, `fxp0`). This convention is chosen because we may have several network interfaces, like `fxp0` and `fxp1`. In summary, there is a directory for configuration of the VIF structure (`/proc/network/`). And, each process entry possesses a `sockets` subdirectory for VIF association (`/proc/[pid]/sockets`). Under that directory are subdirectories for each socket and software links pointing to their respective target VIFs.

IV.1.2.3 Operation Model The operation model is briefly described next. We first show how the system interface is used by applications, and discuss some other issues related to the API.

Low-level Interface For control of the VIF system, we use common file management system calls, such as directory creation, soft linking, file read and write. Since VIFs are represented by files and directories, we may also control the VIF system using corresponding commands, such as `mkdir` and `ln`. Although we do not expect ordinary users to use the file management commands for VIF operation, we use such commands for simplicity of the illustration, below. First, to *create* a VIF, the `mkdir` command (or system call, to be exact) is used under `/proc/network` directory. Likewise, the `ln` command (a system call, to be exact, again) is used to *attach* a VIF to a process or a socket. For example, the operation `ln`

-s /proc/network/fxp0/5 issued in directory /proc/[p3]/sockets/2/ has the effect of attaching the VIF 5 to the second socket of process 3. Analogously, detachment and removal of VIF are possible with rm and rmdir.

In this procfs abstraction, it is easy to support various attributes for a VIF, through parameter files. To set any value to a certain parameter, simply write the ASCII value to a file representing that parameter. To read the values, simply read the file contents.

Flexibility of Functionality A VIF directory is a natural representation of a VIF, in which attributes of a VIF correspond to files in the directory. Because of the file system abstraction, the API is highly flexible.

For example, to realize various types of traffic control and packet scheduling in the control model, we can simply create a file in the VIF directory that keeps the *scheduler type* so that each virtual interface can specify its own packet scheduler (not shown in the figures). In the prototype implementation, users can write a scheduler ID onto the file handle, for specification of the scheduler type.

It is also possible to further extend the interface. For example, we may allow users to inject their own packet scheduler code onto the VIF directory. We may also create a file for packet filtering rules. These approaches are investigated in the following chapters.

Multiple Network Interfaces One complication is a situation where multiple network interfaces are involved. Suppose that a machine has two network interfaces (fxp0 and fxp1) connecting the machine to different networks. A process on the machine might want to use fxp0 heavily, but might need moderate usage of fxp1. In this case, we need to have separate VIF structures for the two interfaces. Note that having two interfaces is not a rare case, since every Unix system has a loopback interface at least, in addition to any physical interface.

In our proposed scheme, each interface has its own VIF structure under /proc/network. (Figure 4.4 shows just one interface for simplicity.) Accordingly, each process has independent attachments to each one of the VIFs, under /proc/[pid]/sockets. It is natural to designate the name of the real interface for the associations to discriminate each independent VIF structure (fxp0, in the shown case).

IV.1.2.4 Remarks The proposed system interface is powerful for the following reasons.

1. The hierarchical structure of the virtual interfaces and its operation are more intuitively represented than a system call based approach.

2. Resource protection is naturally realized with existing file permission semantics, that is, users can only control VIFs if they have proper access rights.

3. The abstraction enables various types of network control in a single consistent interface, for example, work-conserving and non-work-conserving.

4. It supports controls of various types of OS abstractions (or resource units) representable in the process file system, which are not limited to process and sockets.

5. Very importantly, all the above advantages are valid for any programming environment that supports a common file access API, even without a system call interface, such as shell scripts.

The primary drawback of the scheme is its control overhead, compared with a set of dedicated system calls, another possible API. For example, we need to invoke extra system calls, `open()` and `close()`, just for parameter inspection. Accordingly, the overhead is profiled in the evaluation section later to further justify the design decision.

IV.1.3 Sample Applications

Next, sample applications are presented as case studies, to demonstrate the prototype implementation. These applications provide a variety of ways for system administrators and users to control their traffic.

IV.1.3.1 End-User commands for traffic control The first category of the traffic control application is traffic control commands for end users. This category includes extension of shells and the `netnice` command [99], which allow users to dynamically limit bandwidth consumption of applications. For example, a user can specify the bandwidth consumption of a web browser at startup, as follows:

```
% netscape @512Kbps
```

In contrast to traditional per-flow control, users and system administrators can easily control the network resources without knowledge of the underlying protocol, utilizing the simple '@' specifier. Note that such a control is not possible with traditional implementations, because their control model lacks properties needed for the control as discussed in the last chapter.

In addition to the static specification at startup, the end user or the system administrator can dynamically throttle the network usage of running processes, with `netnice` command. Syntax of the arguments is similar to `nice` command of Unix system.

```
% netnice 512Kbps $pid.
```

In contrast to the static configuration of the traditional approaches, this simple command realizes dynamic control of ongoing connections. This is a simple illustration of the advantage of our proposed OS-oriented scheme over traditional implementations with network-oriented control framework designed solely for administrators.

IV.1.3.2 Control of commodity servers The second category of control is traffic control of commodity services, such as ftp and ssh. Traditionally, service specification and resource specification of the services have been handled separately, utilizing different mechanisms. For example, on a Unix system, the specification of server processes is in `/etc/inetd` and the specification of network might be found in `/etc/rc.network`. This is a natural choice in the traditional approach, since the network control is independent of the communicating applications at the end. For example, control of FTP traffic is done by simply limiting

ports 20 and 21 of TCP. Likewise, they may configure the system to do some queuing control over port 22 of TCP to selectively prioritize SSH traffic.

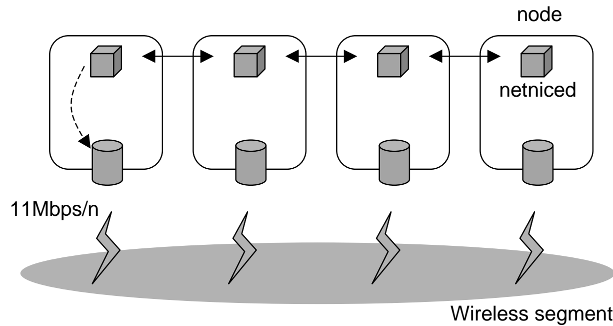
However, since network resource management is an inseparable part of service specification, the separation has limited the resource allocation strategies available to server systems. As a counterpart to the traditional control model, we have proposed the *extended-inetd* model (originally presented in [99]). This scheme wraps each process with a dedicated virtual network interface and dynamically controls the traffic specification of each service. With our implementation, system administrators can specify the allocation of network resources in the service specification file (`/etc/inetd.conf`, in the Unix tradition), as follows.

```
$ cat /etc/inetd.conf
ftp     tcp  /usr/libexec/ftpd     ftpd -l
login   tcp  /usr/libexec/rlogind  rlogind  @32K
telnet  tcp  /usr/libexec/telnetd  telnetd  @32K
$ inetd  @512K
$
```

First, we give 512Kbps limitation to all commodity services handled by `inetd`, simply leaving the rest of the bandwidth to other activities on the system. Second, when interactive traffic arrives at the endhost (for `telnet` and for `rlogin`), it invokes the appropriate server program, and wraps it with a VIF which has 32Kbps bandwidth. Lastly, the rest of the bandwidth is evenly shared by ftp connections for bulk data transfers. Note that the configuration sample does not show all the fields required, for simplicity of presentation.

This is another illustration that suggests the advantage of our proposed scheme over the existing network-oriented approaches.

IV.1.3.3 QoS manager To further explore the advanced traffic control realized with the VIF scheme, we have also implemented a multi-purpose QoS manager [102]. Originally designed for traffic control in load-balancing clusters, the *netnice daemon* realizes variety of end-host oriented traffic control, through *scripting of traffic control algorithms*. The latest version provides a JavaScript-based scripting environment, with a standard class library and event-driven control framework.




```
1 var vif = system.get_root("wi0");
2 var node = new Tuple(1);
3
4 function timer()
5 {
6         vif.bandwidth = 11 * 1024 * 1024 /
7                         node.size();
8 }
```

Figure 4.5: Simple script for fair-sharing of wireless network bandwidth

To illustrate the simplicity and strength of a traffic control script, we give an example for fair-sharing of available bandwidth in a wireless network, below (see Figure 4.5). For this control, we simply deploy `netniced` on each node with the sample script.

The `system` object is an object representing the control system, and `get_root()` is a method that returns a VIF object connected to the physical interface, given an interface name (Line 1). The VIF object instantiates a virtualized network interface, which allows easy control of network traffic. The tuple object is a simple abstraction for inter-daemon communication (Line 2). The object is, conceptually, shared among all the daemons on the segment, and used to share a variable and related statistics. In this example, a tuple object is used to count the number of neighbors on a segment at Line 7. A function, `timer()`, is an event-handler which is automatically called when the QoS manager raises a periodic Timer event, thereby realizing easy polling-based control. At Lines 6 - 7, a fair share of the bandwidth is calculated and the bandwidth parameter of VIF is configured accordingly. Combined together, this short script realizes automatic fair sharing of the wireless bandwidth.

This simple script illustrates the ease and the power of traffic control scripting, based on the VIF model.

IV.2 EXTENSION OF FUNCTIONALITY - PACKET MONITOR

Next, a virtualization scheme of network I/O needs to provide generality and flexibility in the controls it realizes. Accordingly, to demonstrate these properties of the hierarchical virtualization model and the prototype implementation, a packet monitoring mechanism is added to the framework, which would prove extensibility of the scheme.

IV.2.1 Overview - the Port mechanism

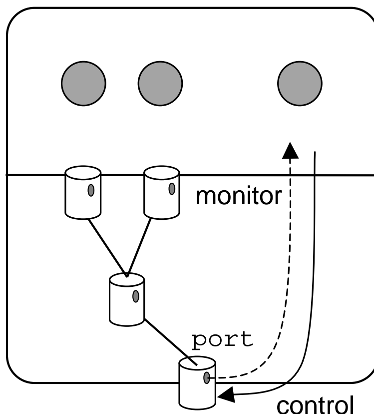


Figure 4.6: Port interface of the VIF system

The monitoring facility, *port*, is used to capture packets that travel through a particular VIF (Figure 4.6). The captured packets are copied and sent to user space, to be used by monitoring applications. Accordingly, the mechanism serves as a peephole into a VIF. It also favors adaptive control of traffic, where traffic is controlled based on monitored information, with flexible control granularity.

To ease development of such a control application, the monitoring API is kept compatible with the de-facto standard for packet capturing, Berkeley Packet Filter (BPF) [88]. Accordingly, system users can use popular network monitor applications, such as `tcpdump` [69] and `ethereal` [34], for monitoring of traffic that goes through the virtualized interfaces. Because a VIF is attachable to terminating entities on the host, such as processes and sockets, the mechanism provides ability to monitor network applications at arbitrary granularity, which has never been realized.

This section first presents the implementation details, followed by the implications of the mechanism. As briefly illustrated, the simple extension of the virtualization model realizes a variety of applications. Quantitative evaluation of the mechanism will be made in the following section, along with performance evaluation of the base system.

IV.2.2 Implementation detail

The port mechanism realizes monitoring of the virtualized interfaces, in a similar way as BPF, which is designed for monitoring of physical interfaces. A technical challenge in the implementation is how to keep compatibility with the BPF mechanism, which is addressed as follows.

First, the interface to tap the traffic is represented as a file, `port`, in its corresponding VIF directory under `/procfs/network`. In addition, to make the port interfaces under `procfs` visible at the standard mount location of BPF (`/dev/bpf*`), we devised a technique, which we call *device stealing*. It uses a soft link from the BPF location to the VIF port, to temporarily "steal" an application's access to the BPF device, as shown below. The procedure is shown as a sequence of user commands, just to deliver the idea.

```
# ls /proc/network/fxp0
bandwidth  recv       type       port
drops      send       weight
# cd /dev
# rm bpf0
# ln -s /proc/network/fxp0/port ./bpf0
```

Second, we extended the packet format of the original BPF, to pass richer information

```
struct bpf_hdr {
        struct timeval bh_tstamp;     /* time stamp */
        u_long bh_caplen;             /* length of captured portion */
        u_long bh_datalen;            /* original length of packet */
        u_short bh_hdrlen;            /* length of bpf header */

        /* Extension */
        u_int bh_pid;                 /* process id */
        u_short bh_direction;         /* direction */
};
```

Figure 4.7: Header Format of the packet filter

to the userland, while providing compatibility to legacy programs. Figure 4.7 illustrates the change: `bh_pid` and `bh_direction` are added at the end of the BPF header. The second field, `bh_direction`, was added because we need to discriminate incoming packets and outgoing packets. Note that the change does not require legacy applications to be recompiled, since we could adjust the size of the structure by changing `bh_hdrlen` (good design practice by the BPF people!).

Third, for BPF compatibility of the `port` abstraction, we changed semantics of some `ioctl()` commands, upon which the BPF control is based. For example, each VIF is already attached to a physical interface, and there is no need for explicit attachment, as is done in the BPF case. Hence, a command to attach a BPF device to a physical interface (`BIOCSETIF`) is simply ignored.

Lastly, we changed the semantics of write operation on the interface, to allow the *packet diverting* operation, which diverts packets into userland. In the original BPF, writing onto an interface causes injection of a raw packet into the network. To support the diverting operation, we slightly modified the `port` interface, so that the packets can be re-inserted into the original flow. For this operation, monitoring applications first block packet forwarding in the target VIF, by applying a non-discriminative packet filter (discard-all), or by changing the type of the VIF to a blocking VIF. Otherwise, each packet would be duplicated, by the re-insertion. Then, the monitoring applications read packets from the `port` file. After

performing certain operations, they can return packet data back to its original flow, just by writing the packet buffer back to the same device file, `/proc/network/fxp0/port`.

Since packet diverting violates the BPF semantics of data writing, the mechanism is not BPF-compatible in a strict sense. Nevertheless, the dominant infrastructure is mostly for packet capturing, and there is only one major program (`rarpd`) that uses BPF writing, to the best of our knowledge. Hence, we believe that this change does not have a significant impact on the compatibility issue.

IV.2.3 Implication of the VIF monitoring capability

Despite its simplicity, the mechanism overcomes many of the deficits of existing monitoring mechanisms, as illustrated below.

IV.2.3.1 Monitoring Granularity First of all, although the end-node mechanism must be capable of monitoring activity of applications on the host, this basic functionality has not been always provided. For example, we may want to detect traffic from a Trojan horse program, which may use the legitimate port 80 of TCP to communicate with intruders waiting outside for a chance to get inside.

However, such an application is not easily detected with existing monitoring mechanisms. For example, BPF does not allow monitoring on a per-application basis, since it is attached to network interfaces, at the bottom of the network system. Because the monitoring granularity of this mechanism is fixed to an interface, it cannot be used for monitoring of applications, or processes.

Our proposed model enables network monitoring of terminating entities at flexible granularity. For example, in Figure 4.3, VIF 1 and VIF 2 can monitor the network I/O of p1 and p2, respectively, while VIF 3 can be used to monitor both p1 and p2 at the same time. VIF 4 and VIF 5 are used to monitor sockets used by threads in p3, and VIF 6 in used for all the processes on the host. This mechanism can also be used for monitoring of packets that pass through a firewall (VIF 6), to confirm functionality of the firewall rules, by monitoring a VIF connected to the root VIF.

This flexibility is not possible with other mechanisms that work solely at the network layer. As illustrated, the flexible granularity of the VIF system is effective in traffic monitoring, as well as in traffic control. Actually, this integration favors a variety of controls, as illustrated next.

IV.2.3.2 Integration of Monitoring and Control Traditionally, packet monitoring mechanisms have been designed solely for monitoring of traffic, and operating systems cannot efficiently control traffic based on the monitored information. Conversely, control mechanisms, such as packet filters and advanced queuing mechanisms, are designed mainly for control of traffic, and rarely provide monitoring facility.

The implication of the dissociation in monitoring and control is twofold. First, as suggested, we cannot control traffic efficiently based on monitored information. This implies that the feedback control is not efficiently realized with existing mechanisms. Second, we cannot monitor packets that passed through the filter. This implies that there is no efficient means to confirm the effect of packet filtering. These facts limit the way we secure the system. For example, if a system needs more data to evaluate a suspicious flow, but do not want to leave the flow open and vulnerable, they might want to slow down the flow temporarily, rather than to drop all packets from that flow. A mechanism is needed to somehow bridge the gap between control mechanisms and monitoring mechanisms on end hosts.

The VIF system, coupled with the monitoring facility, provides a perfect solution to the problem, just by integrating the control mechanism and a monitoring feature. It also realizes integration of network security and QoS, because we can block communication of a VIF by setting bandwidth of the VIF to zero. The integration is beneficial to reduce system complexity. Another benefit is its performance advantage: the integration of security and QoS would reduce overhead by eliminating redundant operations in the network subsystems, such as packet classification.

IV.2.3.3 Possible applications The port mechanism provides compatibility with the dominant monitoring infrastructure, BPF. Accordingly, this allows users to reuse `libpcap` applications for monitoring of application behavior. Also, we can use the diverting mecha-

nism for further control of traffic, for example, modification of packet data. One example is a proxy module, which inspects mail traffic and removes virus-infected email attachments.

Additionally, it offers great programming freedom in traffic monitoring and control. Because the mechanism can be used to inspect network I/O of applications, it allows for stateful analysis of the communication semantics of applications, based on protocol knowledge. Because packet classification is made through the tree-like structure of the VIF system, each monitoring program needs to consider only communication content without being bothered by classification task. This feature greatly simplifies processing in monitoring programs.

All of the advantages above are realized by a slight extension of the VIF model and implementation. Accordingly, the monitoring mechanism inherits all the properties of the VIF system, such as flexible control granularity and access control. This fact suggests extensibility of the model and the implementation, as we claimed.

However, these advantages could be easily offset by poor performance. Hence, efficiency of the implementation is examined, next, to show this is not the case in our prototype.

IV.3 EVALUATION

In this section, the presented implementation is evaluated, both qualitatively and quantitatively. First, we present the traffic pattern generated by the VIF system, and show that it complies with the traffic specifications. Second, we evaluate the control overhead and then the overhead of the packet schedulers is profiled. Third, performance of the added monitoring facility is measured. Lastly, qualitative evaluations are made at three distinct levels, namely, kernel implementation, functionality, and application.

The hardware specification used in the experiments is: an Intel Celeron 2.0Ghz CPU with 512MB of main memory, and a Gigabyte GA-8INXP motherboard (Chipset: E7205). The motherboard has an on-board GbE NIC, the Intel PRO/1000, and we added another to the system, the Intel 82559 Pro/100 Ethernet. We connected the GbE interfaces of the machines with a GbE switch, a D-Link DGS-1008D, for data plane, and 100/10 interfaces with another switch, a D-Link DI-604, for the control plane.

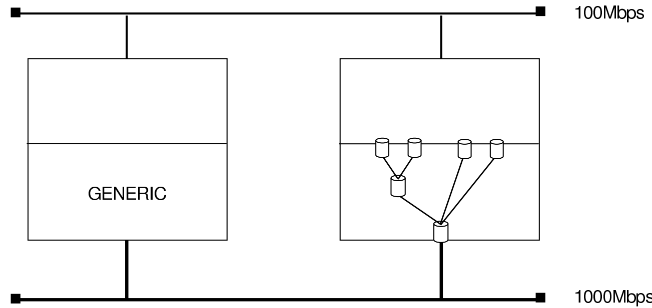


Figure 4.8: Experimental Setting

IV.3.1 Traffic Profile

First, a simple experiment is conducted to verify that traffic generated by the prototype complies with VIF specifications. In the experiment, a modified kernel (NETNICE) and a normal FreeBSD kernel (GENERIC) are installed on the target machines, and traffic is generated between them, as shown in Figure 4.8.

On the NETNICE machine, the VIFs are structured as shown in Figure 4.3. Then, these VIFs are configured as follows. VIF 1 and 2 receive 66% and 33% of VIF 3's bandwidth share, respectively. VIF 3 is limited to 4096Kbps, VIF 4 to 8192Kbps, and VIF 5 to 2048Kbps. The root vif (VIF 6) is configured as 1000Mbps, which is much bigger than the hardware capability, to avoid bottlenecks.

The simulation scenario is as follows. A process attached to VIF 5 continuously sends packets as background traffic. From time 20 to 39 and 30 to 49, VIF 1 and VIF 2 send packets through VIF 3. Additionally, to demonstrate *regulation of input traffic*, the GENERIC machine is configured to send burst data toward a process connected to VIF 4, from time 5 to 14.

We captured the traffic on the GENERIC machine, using the `tcpdump` command, and plotted the throughput (Figure 4.9). It is clear that the implementation regulated the traffic as specified. Traffic for VIFs 3 and 6 are not shown since they are simply sum of VIFs 1+2 and VIFs 3+4+5, respectively.

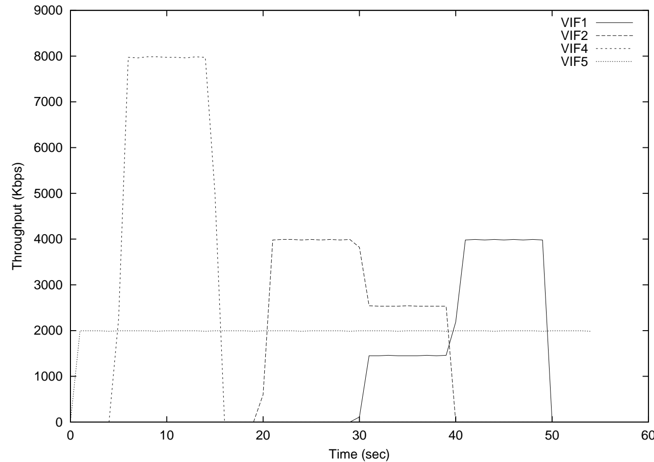


Figure 4.9: Throughput of VIFs for both input (VIF 4) and output (VIFs 1-3, 5) traffic

IV.3.2 Overhead of management operations

Next, overhead of the management operations is measured. In this experiment, the following operations on VIFs are tested: creation, deletion, attachment, detachment, read, and write. Note that read and write require an `open()` system call before the actual I/O takes place. For other operations, a single system call is made. For measurement, the rdtsc counter of the Pentium architecture was used, and 100 runs were made for each measurement.

The results are shown in Table 4.1. Taking the minimum values measured as the representative results, we can conclude that the overhead of management of VIFs is about 28 μsec on average, likely to be justifiable for most applications. The minimum values are used because the values measured by the rdtsc counter include interrupt handling that may occur while carrying out the VIF operation.

IV.3.3 Scheduler Overhead

We also measured the overhead of the VIF scheduling. To this end, we first connect several VIFs in series and measure delay and maximum throughput changing the number of VIFs, to measure the impact of the tree depth. In addition, since work-conserving and non work-conserving VIFs handle scheduling differently, two series of experiments (WC and NWC)

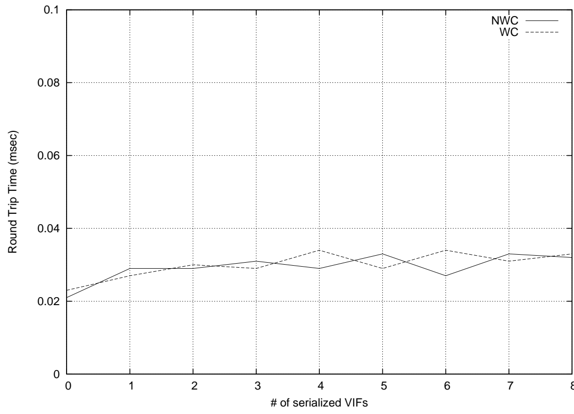
Operation	Min.	Av.	Max.	SD
create	14	17	102	2
delete	39	41	66	2
attach	22	34	56	3
detach	24	25	41	2
read	37	40	145	1
write	30	32	68	4

Table 4.1: System Call Profile (in μsecs).

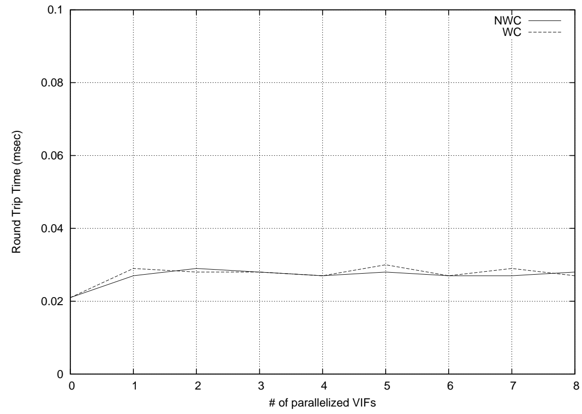
are run for each metric, namely throughput and delay. For the experiments, a loopback interface is used, and overhead for output traffic is measured. This is because it better represents performance of the algorithm when not disturbed by hardware interrupts.

For the measurement of delay, we measured the Round Trip Time (RTT) of `ping` packets, utilizing the loopback interface. In this setting, each packet travels from userland, down through the protocol stack to the loopback interface, and then back up to the socket of original process, without any access to network hardware. Therefore, observed RTT is a linear function of the overhead of the VIF processing. One hundred samples were taken for each round and we used minimum values for further analysis.

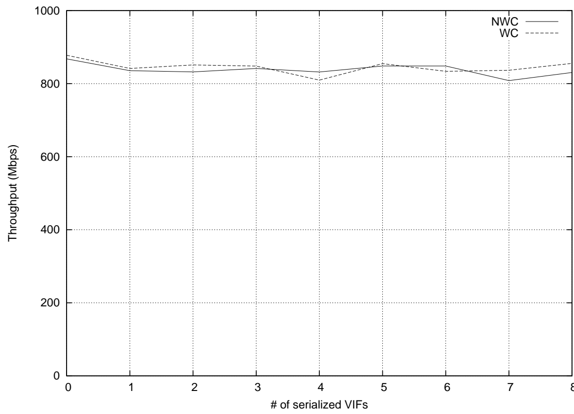
Figure 4.10(a) shows the VIF delay in the serialized configuration. The X axis shows the number of VIFs connected to each other in series (when the number of VIFs is zero, we bypassed the VIF subsystem), and the Y axis is the minimum RTT in milliseconds. We see that there is a 10 μsec jump between 0 VIF and 1 VIF configurations, but less penalty we once entered into the VIF subsystem (i.e., for number of VIFs ≥ 1). Although the relative overhead looks severe in the graph, the absolute delay is negligible for physical interfaces where data transfer takes much longer. We also see that the difference between a work-conserving VIF and a non-work-conserving VIF is not significant, although they utilize different algorithms.



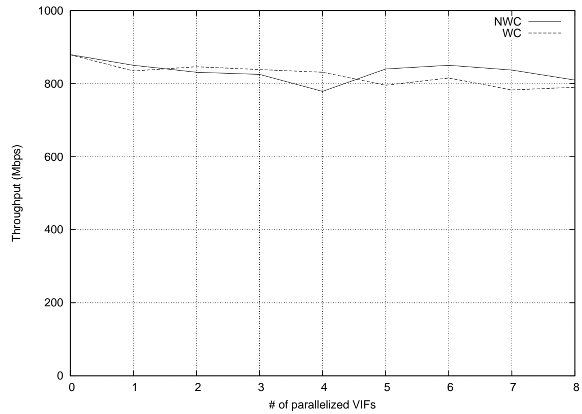
(a) VIF delay in a serialized setting



(b) VIF delay in a parallelized setting



(c) VIF throughput in a serialized setting



(d) VIF throughput in a parallelized setting

Figure 4.10: Scheduler overhead

For throughput, we wrote a program that generates UDP traffic and receives the packets, measuring its bit rate. We configured the program to use the loopback interface so that it can measure the peak performance inside the system without any interrupts and context switches. One thousand samples were taken for each configuration and we used the maximum value for further analysis.

Figure 4.10(c) shows the throughput of a VIF. The X axis is again the number of VIFs in series (again, at #VIFs = 0, we bypassed the VIF subsystem), and the Y axis is the maximum throughput. As shown, we see almost no slowdown in the max throughput as we increase the number of VIFs. Again, the difference due to the scheduler type is not significant.

Additionally, to know the overhead for parallel settings of VIFs, we measured the delay and throughput for VIFs configured in parallel. In the experiments, starting from one VIF connected to a root VIF, we add one more VIF connected directly to the root VIF in each turn, increasing parallelism up to eight VIFs. We measured the delay in the same way as the serialized setting. For throughput, we configured the traffic generator to open multiple sockets, each connected to dedicated VIFs.

Figure 4.10(b) and Figure 4.10(d) show the VIF delay and throughput for the parallel configuration. We observe no significant performance penalty in terms of delay, as we increase the number of parallel interfaces, except for the jump between 0 and 1. Regarding throughput, we see the average performance is not severely affected by the number of VIFs. In this graph, again, the difference between non-work-conserving VIFs and work-conserving VIFs is not significant, although they use different queuing strategies.

IV.3.4 Port mechanism

To profile the port mechanism, a microscopic study and a macroscopic study are conducted. The former is intended to measure microscopic behavior of the implementation for each packet processing. The latter is to measure the macroscopic behavior of the port mechanism, including throughput and overhead of the VIF processing. In these experiments, we

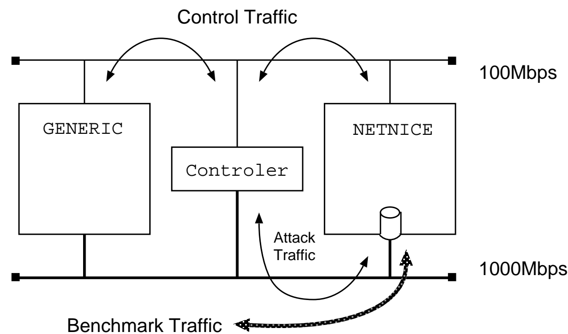


Figure 4.11: Experimental Setting

compared the implementation with the de-facto standard packet capturer, BPF.

IV.3.4.1 Capturing performance The hardware setting used in the experiments are shown in Figure 4.11.

For the microscopic measurement, we again used the rdtsc clock counter and took 10,000 samples each, measuring the number of CPU cycles needed to process each packet. We prepared three scenarios: (a) Accept, (b) Host match, and (c) Host miss. In the first scenario, all packets generated are accepted, utilizing a filter rule which matches every packet. In the Host match case, all generated traffic was to a single destination (say 192.9.200.1), and we used a rule which matches all the packets ("accept dest 192.9.200.1"), resulting in the acceptance of all the traffic. In the last case, Host miss, packets are generated to an unused port on the host, and none of the traffic matched a rule, dropping all the traffic. For BPF, we measured cycles to execute a tapping routine. For the port mechanism (shown as NPF), we measured cycles to execute a corresponding routine for packet capturing. We used the minimum values for analysis. A capturing process was run with a real time priority, to avoid packet dropping by the listener.

Figure 4.12 shows the result. We observe the advantage of the port mechanism (NPF) in all the cases. The port mechanism internally translates BPF filter code provided, into a native binary, although the BPF micro-machine interprets the code iteratively. The better performance is due to the binary translation. The significant difference between the accept and reject packets cases (i.e., first two vs. last scenarios) is due to the fact that the packets

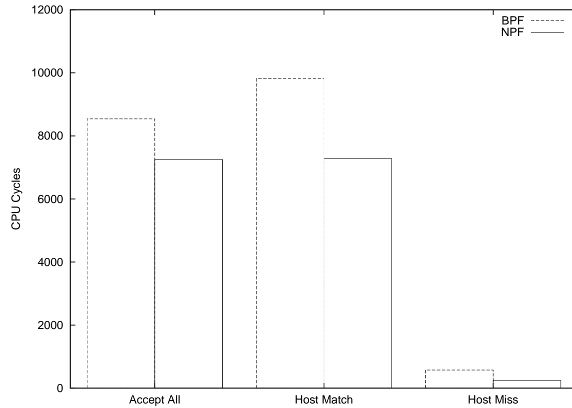


Figure 4.12: Packet Capturing Performance of BPF and the port

are copied to userland if they are accepted.

IV.3.4.2 Impact on Throughput of mainstream traffic Next, we assessed the impact of the implementations on the throughput of mainstream traffic, as a macroscopic measurement. We used `nttcp` [12], a traffic benchmark program, while running a monitoring process on the traffic.

Figure 4.13 shows the result. In the Accept All case and Host Match case, BPF exhibited better performance. We did not observe any difference in the Host Miss case. As shown, the advantage we observed in the microscopic metric was offset by the VIF processing overhead, resulting in overall similar performance for both schemes.

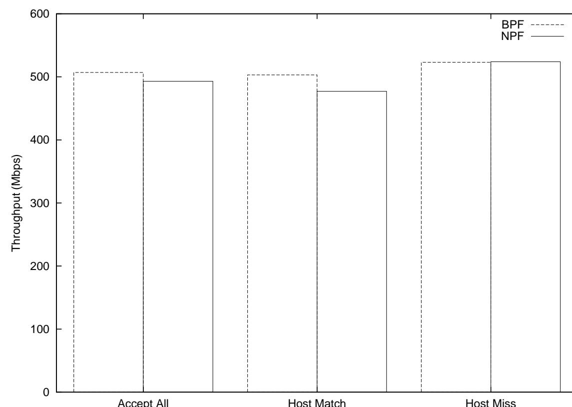


Figure 4.13: End-to-End throughput while being captured

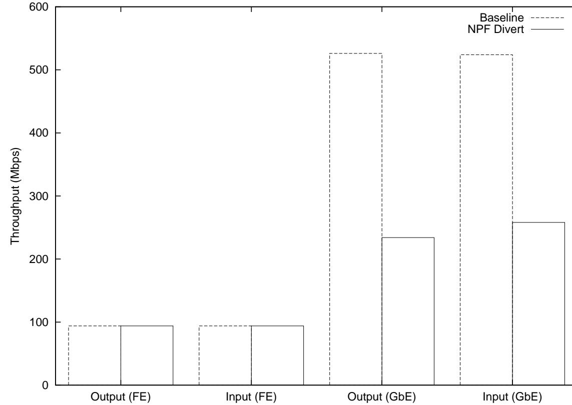


Figure 4.14: End-to-End throughput of Packet Diverting

IV.3.4.3 Packet Diverting Lastly, we measured performance of the diverting operation. We again used `nttcp` on a diverting VIF, and measured the throughput for both directions, input traffic and output traffic. Additionally, in this experiment, we tested both Fast Ethernet (FE) speed and Gigabit Ethernet (GbE) speed, to see how it scales. Again, to avoid packet dropping, we run the listening (diverting) process with a real time priority.

Figure 4.14 shows the result. Baseline is for the run without the diverting operation. As shown in the four bars on the left, we did not observe significant difference between the modes in the Fast Ethernet setting. This is because the Celeron CPU was fast enough to divert all the packets at the link speed.

The right half of the figure is for the Gigabit Ethernet setting. In this case, we observed significant slowdown in the diverting mode. The peak performance of the baseline cases were 526Mbps and 524Mbps, for output traffic and for input traffic, respectively, while the divert performance reached 236Mbps (45% of Baseline) and 258Mbps (49%). Because the diverting operation doubles data transfer between user space and the kernel, the result suggests that the buffer copy dominates the packet processing. It is obvious that this mode should only be used when it is worth the benefits of diverting packets with such performance degradation. Another approach is to avoid the costly buffer copies by the zero-copy approaches [32], but the exploration of such mechanisms is left for future work.

IV.3.5 Qualitative evaluation

Lastly, a qualitative evaluation of the prototype is made in three distinct levels: kernel implementation, functionality, and application.

IV.3.5.1 Kernel implementation level The prototype implementation exhibited the following properties.

1. The prototype implementation on FreeBSD4 was successfully ported onto FreeBSD 5, OpenBSD, NetBSD, and Linux. Note that FreeBSD 5 kernel significantly differs from FreeBSD 4, on which the prototype was first developed. Linux also has a totally different kernel, compared with FreeBSD 4. Accordingly, the hierarchical virtualization model was proved to be implementable on ordinary modern operating systems. All of the kernel patches are publicly released as open source software.
2. It also proved that the mechanism is easily extended to support new communication protocols (such as IPv6), and new network interface cards. For example, we confirmed that a new Gigabit Ethernet card works without any system modification. This is because the mechanism works between the interface layer and the network layer, and does not modify device drivers. This property is sometime missing in the traditional implementations for network control, like ALTQ [31].
3. The prototype demonstrated that *early demultiplexing* can serve different purposes from those presented elsewhere. Faithful implementation of protocol layering at end-hosts is becoming obsolete for system security and I/O controllability. For example, [92, 40] has shown that early demultiplexing protects CPU cycles from malicious network interrupts, although it violates the customary layering rules. The hierarchical virtual network interface approach serves as another illustration that justifies early demultiplexing on modern operating systems.
4. The prototype implementation demonstrated flexible extension of VIF functionality, as shown through the traffic monitoring scheme. However, although the model covers a broad spectrum of functionalities and the implementation can accommodate flexible extension of the functionalities as demonstrated, it also revealed that new functionality

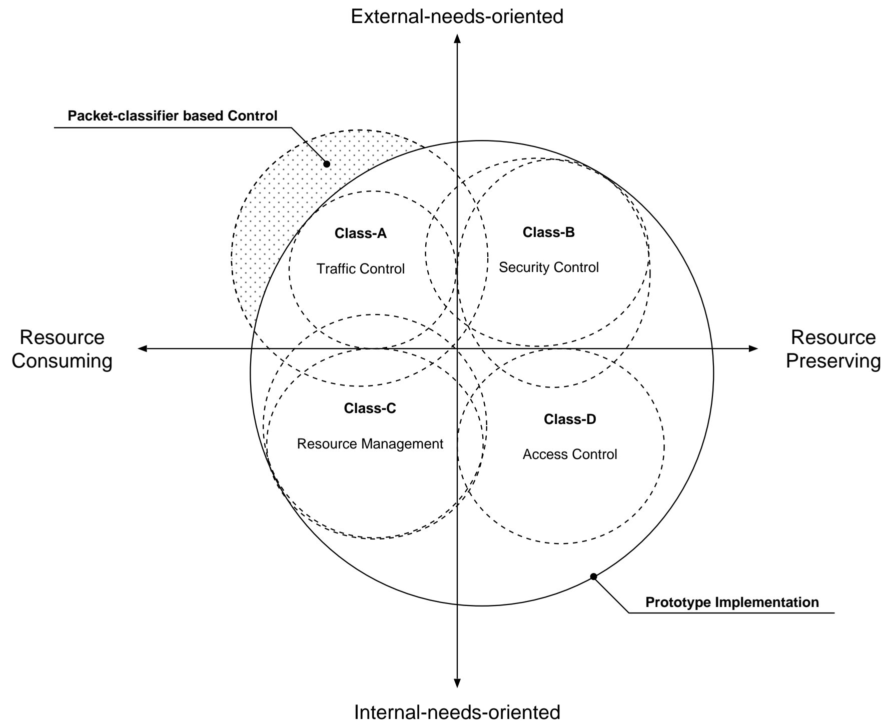


Figure 4.15: Functional Coverage of the prototype implementation

must be hardcoded in each operating system. Accordingly, it would be preferable to have a mechanism that allows flexible extension of VIF functionality so that it can be portably extended.

IV.3.5.2 Functionality level The functional coverage of the prototype implementation is shown in Figure 4.15, and summarized as follows.

1. The prototype realized the hierarchical structuring of virtualized interfaces, allowing users to attach VIFs to OS-supported entities at any desired granularity, such as a process group, a process, or a socket.

2. It supported various traffic control features for QoS control, such as bandwidth control, fair queuing and priority queuing in a unified manner.

3. It realized partitioning of resource usage, while providing a resource management model.

4. It integrated traffic control for QoS and security control in a single framework.

All of these features correspond to what the control model intended to offer, and the prototype successfully implemented the hierarchical virtualization model, in an efficient manner.

However, the prototype is still incomplete in that it does not support control by address-port pairs, although the traditional control model supports such a control. Accordingly, functional coverage of the primitive is not perfect, and thus, such a control must be supported on the virtualization framework, to claim completeness of the proposed control model.

Additionally, the hierarchical virtual network interface model presented in this paper does not provide guaranteed network bandwidth, as pursued in former studies [59], although it does provide upper-bound regulation. The lower bound guarantee is not addressed in the discussion, because of three reasons. First, QoS guarantee of end-to-end communication is possible only when the network architecture supports network-wide QoS, which is not yet realized. Second, guaranteed service requires integration of the network I/O and CPU scheduling, which compromises modularity of the proposed system. In other words, packet schedulers and upper-bound regulation systems can be implemented in a modular manner. Third, it requires a specification mechanism for resource usage, which is far beyond the scope of the dissertation. Accordingly, this topic is left as future work.

IV.3.5.3 Application level We have introduced three applications utilizing the hierarchical virtualization system, namely the `netnice` command, the netnice daemon, and extended `inetd`. They controlled network applications and servers without modifying them, and thus, they were a clear illustration of the "retrofitting" property of the OS service. It is also possible to rewrite network applications, such as web servers and streaming servers, to directly use the network control service. Although we have not explored the topic sufficiently, we state several observations we have made, about the traffic control service for such

applications.

1. One of the most noteworthy advantages of end-host control comes from its stateful nature. Consider a server for Electronic Commerce, where customers who have started to choose items have a higher priority than customers at checkout. Implementation of this policy at intermediate nodes would require stateful inspection of every flow and semantic knowledge of different types of payload. End-host control can exploit the local information of each transaction for efficient stateful control, which significantly reduces the processing overhead. This would hold for any process that has several connections and wants to differentiate them. Another example is a web server that wants to give priority to HTML documents over large data transfer for pictures and files.

2. Another advantage regards traffic aggregation. For example, RTSP [115] utilizes a connection for data streaming and another for a health check of the stream. Because intermediate nodes cannot easily recognize the different flows as being bundled, only the originator of these flows would have enough information to perform proper control over the session. Our end-host oriented network control facilitates this.

3. Lastly, our approach can provide simple receiver-side bandwidth regulation, for flow controlling protocols, such as TCP. As we demonstrated (see VIF 4 in Figure 4.9), the sender of TCP traffic throttles its flow to conform to the bottleneck bandwidth, if we limit the rate of a VIF at receiver process. This is because the arrival of the incoming packets is paced by the VIF, acknowledgments for each packet (or groups of packets) are delayed, making the traffic compliant with the bandwidth specification.

Some of these controls can be realized with the traditional packet classifier based approaches. We also admit that traffic control by end-nodes cannot solve all the problems inside the network. However, control at the end-hosts and inside the network are orthogonal, by nature. Accordingly, it is inefficient to overload exceptional operations on the traditional approaches in an ad-hoc manner, if such operations are elegantly addressed by the end-host OS approach.

IV.4 SUMMARY - "HIERARCHICAL VIRTUALIZATION CAN BE EFFICIENT"

This chapter presented a prototype implementation of the hierarchical virtualization model of network I/O.

First, an implementation overview of the prototype is presented. This is followed by case studies of sample applications. Then, the prototype is extended to support packet monitoring of VIFs, as a demonstration of the extensibility of the prototype. Lastly, implementations are evaluated quantitatively and qualitatively.

The evaluation confirmed that the hierarchical virtualization model can be actually implemented on general purpose operating systems in an efficient manner. The prototype realized hierarchical structuring of virtualized interfaces, allowing users to attach VIFs to OS-supported entities at desired granularity, such as a process group, a process, or a socket. Second, it supported various traffic control features for QoS control, such as bandwidth control, fair queuing and priority queuing, in a unified manner. Third, the mechanism realized partitioning of resource usage, while providing a resource management model. Fourth, it integrated traffic control for QoS and security control in a single framework.

On the other hand, the prototype exhibited some limitations. First, functional coverage of the control primitive was not perfect. An illustration is the control by address-port pairs. Second, although the model covers a broad spectrum of functionalities and the implementation can accommodate flexible extension of functionality, a new functionality must be hardcoded in the operating system kernel. These limitations are addressed in the following chapters.

V LIMITED VIRTUALIZATION OF PACKET PROCESSING CODE

The last chapter presented a prototype implementation of the hierarchical virtualization model. The prototype exhibited most of the expected advantages. However, it was incomplete in its functional coverage and extensibility of functionality. An illustration of such shortcoming is traffic control based on address-port pairs.

To overcome the restrictions, this chapter investigates *virtualization of packet processing code*, in the aim of solving both the problems at a time. For this purpose, a code execution environment for in-kernel packet filtering is implemented, on top of the virtualization framework.

The *filter* mechanism is a built-in packet filter of each VIF, based on a micro-machine architecture. System users can place filter code onto their filters, and the filter machines execute the filter code to perform packet classification and filtering. The mechanism is also used for traffic statistics and accounting purpose.

The chapter first presents an overview of the filter mechanism, followed by implementation details. Then, evaluation is made both quantitatively and qualitatively. The mechanism is expected to prove that the hierarchical virtualization model can perform even packet-classifier based control, for example, traffic control by address-port pairs.

V.1 THE FILTER MECHANISM - OVERVIEW

An overview of the packet filter mechanism is shown in Figure 5.1. In this scheme, filter rules are first described in a high-level language, which is *compiled* into filter code with proper optimization. Then, the filter code is passed to the packet filter micro-machine, embedded

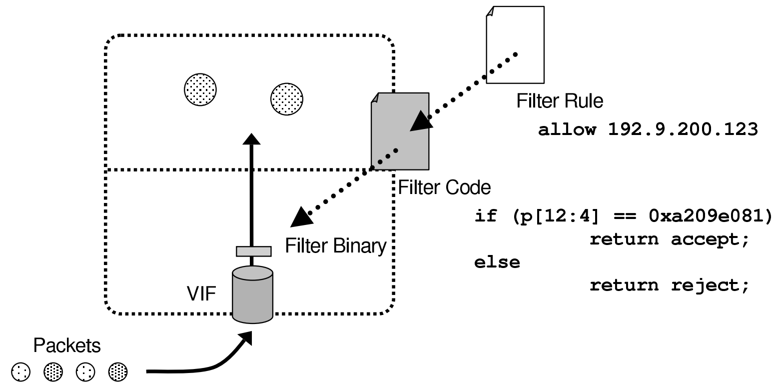


Figure 5.1: Code generation and Instrumentation of filter binary in the kernel

in each of the VIFs, and is translated into native binary code that corresponds to the filter rule.

For an Instruction Set Architecture (ISA) of the filter machine, the BPF micro-machine architecture [88] is reused. The BPF micro-machine architecture is a distinctive feature of the packet filter mechanism, designed specifically for packet classification. In this scheme, a high-level description of packet matching rules is compiled in userland, and optimized packet matching code, expressed in the virtual ISA, is passed to the filter machine. The micro-machine has an accumulator and a scratch memory space. Also, read access to packet payload is provided, with simple conditional jump instructions for flow control of the execution. This architecture has been used for efficient classification of packets [88].

Although the compiler-based rule optimization significantly reduces cycles needed for filter processing of each packet, it introduces overhead in the emulation of the micro-machine architecture. Hence, to avoid the interpreter overhead, further optimization is made by dynamically generating native binaries for each filter code (Just-In-Time compilation). When a filter code written in BPF format is passed to the filter mechanism on a target VIF, the machine translates it to corresponding binary code on the fly to accelerate the filter processing in future. The filter binary is then attached to the VIF, and a packet scheduler of the VIF calls the binary program when a packet arrives. Then, the code determines if a packet it receives needs to be forwarded, or discarded. For statistics and accounting, the code simply uses the scratch memory as a persistent memory space.

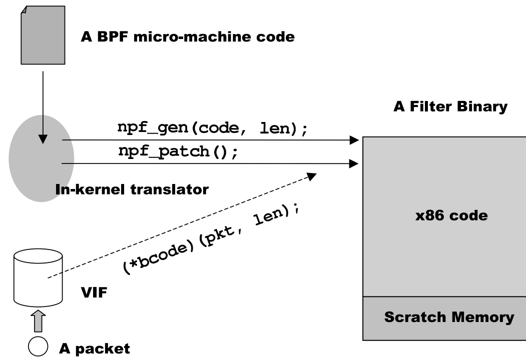


Figure 5.2: Generation and invocation of binary filter code

The advantage of our scheme is fourfold. First, we can optimize the rule matching algorithm in the compilation process. Second, this mechanism is amenable to generating even more efficient code by the binary translation. Third, code safety is guaranteed by the simple instruction set. The programs can access only packet payloads and the scratch memory. Infinite loops are prohibited by not allowing backward branch. Fourth, because we attach the filters to VIFs, the filtering service can be used at any granularity and at any location. For example, we may apply the filter just before a packet monitor, allowing the monitor to inspect the packet which passed through the firewall, a capability of which no other firewall implementation possesses. Note that when applied to the VIF connected to a physical interface (VIF 6 of Figure 4.3), it serves as a traditional firewall. This mechanism can also be used to control the network I/O of processes, by setting local policies of network usage to a VIF directly connected to a process.

V.2 IMPLEMENTATION

The filtering mechanism is implemented on top of the hierarchical virtualization system described in the last chapter, as follows.

First, we modified the NNFS file system interface, so that we can put the filter code onto each VIF. From a users' perspective, the filtering mechanism of a VIF is activated simply by

writing filter code (written in BPF binary form) to a `filter` file in a VIF directory under `/proc/network` that corresponds to a VIF of interest. They may utilize configuration tools to simplify the specification.

Second, we installed the BPF filter machine on each VIF, and modified the packet scheduling routine of the VIF to call the filter machine when a packet arrives. The filter machine is called just like another function, which takes a packet data structure as an argument, and returns positive value if accepted, or zero, otherwise. However, the filter machine causes interpretation overhead, and thus, a binary translator is implemented.

The binary translator is a simple table-driven JIT, and converts a filter program in BPF micro-code into x86 native binary (Figure 5.2). The translator function, `npf_gen()`, first converts the original code into native instructions on allocated memory space. However, target addresses of forward branches cannot be determined in the first path. Because of this, `npf_patch()` is called next, to back-patch unresolved branch targets in the filter code. The translation happens when the filter code is passed to the kernel, using the NNFS user interface described above. It is easy to observe that packet filtering is extremely simplified in this scheme, one function call to a native filter binary (dashed arrow in Figure 5.2), compared with iterative rule-chain matching in ordinary packet filters.

The memory allocation strategy of the code generation is simple. The translator allocates the memory with a roughly estimated size first, and, if the translation process fails due to underestimation, simply discards everything and starts all over with another memory space of doubled size. The scratch memory is in the same memory space, and is kept throughout the life-cycle of the generated binary as a persistent memory. This memory is accessible from user applications, by reading the `filter` file in the VIF directory, and the data is used for statistics and for accounting purposes.

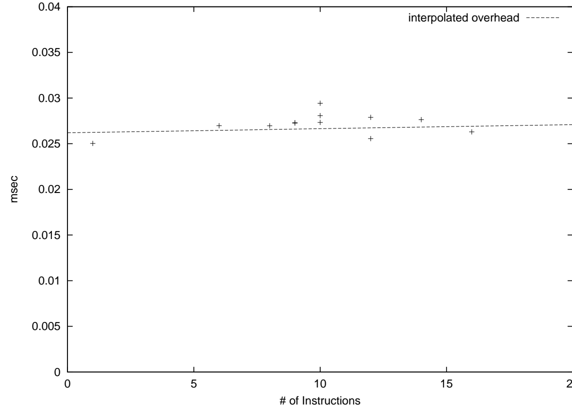


Figure 5.3: Translation Overhead

V.3 EVALUATION

V.3.1 Quantitative evaluation

In this section, performance of the filter mechanism is evaluated. First, overhead for filter instrumentation is measured. It is the time to translate instructions in the BPF machine language into native binary. Second, *microscopic* filtering performance is measured, by counting CPU cycles needed for processing each packet. Third, for *macroscopic* measurement of the performance, performance impact on end-to-end throughput is measured. In the last two experiments, the implementation is compared with a popular software firewall implementation available on the platform, the IP Firewall [6].

The hardware setting used in the experiments is the same as the experiment shown in Figure 4.11, including the hardware specification. We used FreeBSD 4.9 as a base system, and used the GENERIC kernel on the traffic source and a kernel with NETNICE patch on the target host.

V.3.1.1 Translation overhead To measure the instrumentation overhead, we measured latency for a sequence of system calls to install filter code; this requires `open()`, `write()`, and `close()` operations to a `filter` file in a VIF directory. By this sequence of system calls, the filter code is translated into native binary form, and installed onto the target VIF. We

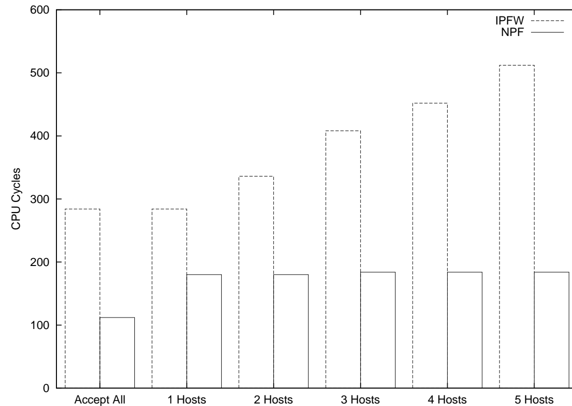


Figure 5.4: Packet Filtering Performance of IPFW and the filter in CPU cycles

prepared several filter rules with different sizes, and plotted the latency of code generation against the number of instructions in the code. For this measurement, the rdtsc counter of the Pentium processor was used, and 100 samples were taken for each filter rule. The minimum value in the run was used for further analysis, because other samples could be affected by interrupts.

The result is shown in Figure 5.3. As shown, there is a linear growth in the latency, depending on the number of instructions in the filter code. However, the slope is quite flat. Each additional host entry in the rule set added just two instructions (compare and conditional jump), and thus, the code generator would scale well for most practical rule sets. The figure indicates that the minimum latency is approximately 25μsec, which can be justified for most network applications.

V.3.1.2 Filtering performance - microscopic Next, microscopic performance of the filter mechanism and IP Firewall (`ipfw`) are measured. For `ipfw`, we measured the latency of its filtering routine in `ip_input()`. For the filter, we measured the corresponding code segment, which calls the filter binary. We counted CPU cycles, which are not easily influenced by system factors, such as CPU clock rate, application protocol, system load, etc. To clarify the difference of the two implementations, several runs were made, changing the number of filter rules for host address matching. We used the minimum values, as their representatives.

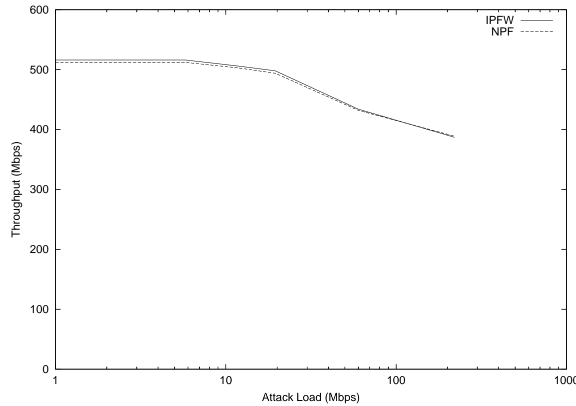


Figure 5.5: End-to-End throughput with attack traffic

The results are shown in Figure 5.4. It is easy to observe the linear growth in the cost for `ipfw`, as the number of the rules increases. On the other hand, performance of the filter mechanism remains almost constant, regardless of the number of rules applied. This owes to the rule optimizer of the scheme.

V.3.1.3 Filtering performance - macroscopic Lastly, application layer throughput under attack is measured to evaluate overall behavior of the implementation. For this purpose, we run a traffic generator on the controller host and generated traffic destined for an application on the NETNICE host, changing its intensity. On the NETNICE host, `nttcp` [12], a traffic benchmark program is run, and application layer throughput between NETNICE and GENERIC is measured.

The result is shown in Figure 5.5. In this setting, the filter did not outperform `ipfw`, but both implementations exhibited similar performance.

V.3.1.4 Discussion In the measurements of throughput, we observed that performance of the filter is equivalent to, or slightly less than, performance of the existing mechanism. There are mainly two factors, which favor our proposed mechanism.

- The structured nature of the VIF scheme is a source of performance gain: We can

selectively apply the *filter* mechanism to specific VIFs, thereby avoiding classification of *all* packets, needed by other mechanisms based on packet classifiers. For example, to monitor traffic for HTTP, we simply wrap a web browser and tap every packet on the VIF, although, in the traditional mechanisms, we need to apply packet classifiers to selectively monitor the packets destined for port 80 of TCP. In the filter case, we simply do not see unnecessary packets, since they are not delivered to the VIF that we control.

- The rule optimization and dynamic code generation outperform the traditional packet filters which processes filter rules one by one. For example, IP Firewall takes filter rules as a *rule chain* written in a high-level description format. The mechanism keeps the rules in the presented form, and for each packet, it iteratively calls functions for rule matching. The filter micro-machine approach avoids such an overhead, by rule optimization and dynamic code generation.

Considering the performance advantage in microscopic measurement, there are two possible reasons, for the degradation in the overall performance.

- The packet filter processing might contribute only a limited portion to the overall cost of communication processing. This is a reasonable assumption because communication processing requires buffer copy and checksum calculation, which are more costly than the simple filtering operations.
- Overhead incurred by the VIF system might have contributed to the overall performance overhead, even though the filter processing is efficient.

Note that the overhead of the VIF system includes the cost for queuing control and bandwidth management, which are not included in the case of IP Firewall. Accordingly, we underestimate its performance unless the IP Firewall is coupled with queuing and bandwidth control modules. We omitted comparison with such cases to avoid controversy about fairness of the experiments.

V.3.2 Qualitative evaluation

For qualitative evaluation, achievements and limitations are summarized first, followed by discussion on implication of the filter system as a security mechanism.

V.3.2.1 Achievements The filter micro-machine is advantageous as a virtualization scheme of network I/O code, in the following aspects.

- Because the filtering can be performed at any VIF, the mechanism possesses the same *flexible control granularity* that the base VIF system has. If the filtering is applied to a root VIF, directly connected to physical interfaces (VIF 6 of Figure 4.3), it would serve as a conventional firewall. If applied to a VIF for a process, it can serve as a private firewall for the process. Existing mechanisms are tightly bound to physical interfaces, and thus, they do not have this flexibility in the control granularity.
- Because filter codes are written in an architecture-neutral instruction set, the code is highly *portable* to any architecture. Further, because the filter machine does not have tight association with underlying operating system, the execution environment is easily ported to new operating system. Accordingly, the filter code has OS independence.
- Kernel internals are hidden from the executed programs and kernel data structures are secured. This *information hiding* is possible because the filter machine supports just a limited set of instructions, and the instructions do not read and modify anything but the packet buffer and the scratch memory.

V.3.2.2 Limitation On the other hand, the filter micro-machine has a limitation in the functionality it can realize.

- The computational model is too simple to efficiently implement data structures, such as a stack or a list. Consequently, it cannot perform complex tasks, such as semantic analysis of traffic. This restricts functionality of the filter, because current software firewalls support filtering rules based on session status of connections. For example, they accept traffic to a certain port only after an appropriate connection is established. It would require redesigning of the filter machine architecture and the rule compiler, to support such an advanced filtering.
- The computational model is not powerful enough to perform sophisticated packet processing, such as queuing control, although it has sufficient descriptive power for simple

filtering of packets. This is mainly because the computational model cannot handle multiple packets.

V.3.2.3 Implication as a security mechanism

Network I/O of the end-host operating system is in between the machine and the outside world. Accordingly, it is a critical strategic point to protect the system from incoming network threats. At the same time, it is a point for the system to audit and control outgoing data flow of the system. These observations suggest that control mechanisms of network I/O on end-host operating systems must possess enough flexibility to provide various security controls, as well as traffic control features, in a unified manner. This justifies the integration of the VIF model and the packet filtering capability. However, the integration has a more positive implication, which is described as follows [101].

Attack trends in the global network are rapidly invalidating conventional firewall technology [14]. For example, a secure network segment is easily threatened by a virus-infected node within the trusted network; cryptography technology has increased the difficulty in packet inspection at the point of border defense; port-forwarding might easily penetrate even the hardest firewall ever built, if the firewall considers only types of application protocols. All of these defiant situations happened because the security control is made just at the network border. Accordingly, there emerged greater needs for detailed inspection and control of traffic *at each endpoint of communication*. This concept is called a *distributed firewall*.

A distributed firewall system typically comprises a network-wide top-down framework for policy management, and an enforcement mechanism at each end-node system [68, 30]. In the same context of network security, but from a bottom-up perspective, are a class of software that allows different software firewalls to cooperate [85], such as the Packet Filter (PF) of OpenBSD [64], the `ipfw` mechanism on FreeBSD [6], IPchains [114], and IPfilter [111] for Linux. However, because they work at intermediate location in the communication, they are susceptible to fraud by malicious applications which mimic regular traffic. We need to monitor the endpoint of communication.

For this reason, system security has been moving toward application oriented control.

For example, personal firewalls on Windows [35, 82, 84] monitor network I/O of applications. Most advanced controls even focus on system call activities [98, 107]. However, these security mechanisms have tight association with the underlying operating system, and thus, their security rules are not appropriate for network-wide deployment, contradicting to the need for distributed security solution.

In sum, a network-wide scheme cannot control applications, and mechanisms for controlling applications cannot be deployed network-wide. Accordingly, there is a mismatch here. The hierarchical virtualization approach, along with the filtering and monitoring extension, bridges the mismatch. It realizes detailed monitoring of a network application, while keeping OS-independence, as exemplified throughout this dissertation. Coupled with a network-wide policy management framework, this mechanism could serve as a building block for a distributed security solution, although a proposal for a complete solution is left for future work.

V.4 SUMMARY - "IT IS LIMITED"

The hierarchical virtualization model and its prototype implementation was incomplete in its functional coverage and extensibility of functionality. To overcome these restrictions, this chapter investigated virtualization of packet processing code. For this purpose, a code execution environment for in-kernel packet filtering and monitoring was implemented, on top of the prototype system.

The *filter* mechanism is a micro-machine attached to each VIF that executes user-supplied filter code on the VIF. System users can place variety of filter codes onto their filter machines to perform packet filtering operation. Thanks to the flexibility of the instruction set architecture, the micro-machine can also execute programs for traffic statistics and accounting purposes. To boost performance, a Just-In-Time compiler is used to generate of optimized binaries on the fly, and the implemented system is evaluated quantitatively and qualitatively.

Systematic profiling illustrated that performance of the implementation exceeds hard-

coded implementations, microscopically. Meanwhile, the macroscopic profiling of filter execution did not show a significant advantage for the proposed mechanism. This suggests that the filter processing occupies just a limited portion of the entire packet processing, and/or overhead of the packet handling in the VIF system is not negligible.

The implemented mechanism performed filtering operations based on address-port pairs, which proved completeness of the hierarchical virtualization model and functional flexibility of the implementation. Further, the qualitative evaluation suggested flexibility in the control granularity, portability of the packet processing code, and appropriate information hiding of the scheme. On the other hand, although the computational model of the filter architecture has sufficient descriptive power for simple filtering of packets, it is not strong enough to perform sophisticated packet processing. For example, because it cannot efficiently implement data structures, such as a stack or a list, semantics analysis of on-going connections is hardly possible. Because the computational model cannot handle multiple packets, queuing control is not possible. Consequently, the filter micro-machine model has limited use as a virtualization model of packet processing code, and we need further flexibility.

VI PARTIAL VIRTUALIZATION OF PACKET PROCESSING CODE

The last chapter presented a simple code virtualization scheme, for packet filters. However, application of the model was limited. Accordingly, further virtualization of packet processing code is needed to demonstrate flexibility of the hierarchical virtualization of network interface model.

For this purpose, we designed another code execution framework, named the *VIFlet* model, utilizing the *embedded Java technology*. In this framework, each virtualized interface possesses a lightweight Java Virtual Machine, and executes user-supplied packet processing Java code customized for their needs.

This organization allows each user to flexibly extend their VIF functionality, utilizing the standardized Java technology. In the prototype system, an in-kernel Just-In-Time compiler was also implemented to further boost the system performance.

This chapter first outlines the design and prototype implementation of the VIFlet scheme. A performance profile of the prototype implementation is described next, followed by qualitative evaluation. The systematic evaluation illustrates that such a kernel extensibility can be realized at practical performance overhead, comparable to hardcoded packet processing code. Our open-source prototype also contributed to investigate protection and optimization approaches of user-supplied packet processing code in the kernel.

VI.1 IN-KERNEL VIRTUAL MACHINE FOR PACKET PROCESSING

This section presents the VIFlet model, a packet processing environment based on embedded Java technology.

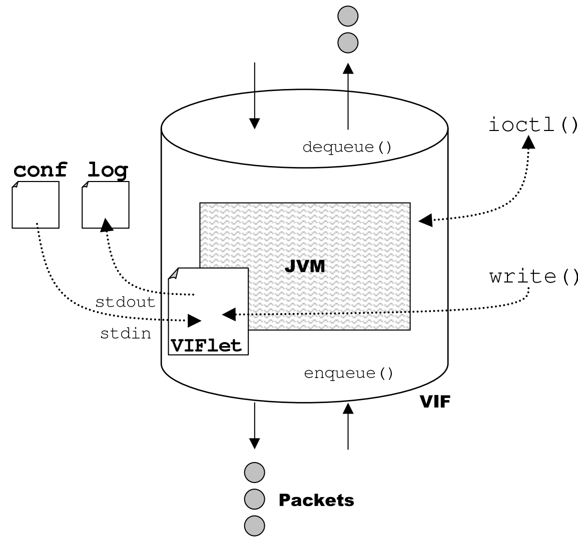


Figure 6.1: VIFlet model

VI.1.1 VIFlet model - Overview

As the name suggests, a VIFlet is a software component that contains Java methods to perform processing of packets. A Java Virtual Machine, embedded on each VIF, executes the methods, while providing necessary protection and isolation of each execution. In order to extend the functionality of a VIF, users can simply program a VIFlet, inheriting the `VIFlet` class provided in the class library, with any ordinary Java development environment. Then, they can safely run the software component on their VIFs to customize the virtualized network interfaces.

An overview of the VIFlet model is presented in Figure 6.1. In this model, a Java Virtual Machine (JVM) is embedded in each VIF (a cylinder), which runs a VIFlet class. To illustrate other parts, such as `enqueue()` and `dequeue()`, sample code is shown in Figure 6.2. The example, PriQVIFlet, is a VIFlet which does simple priority queuing, which works as follows.

First, we declare a VIFlet by inheriting the `VIFlet` class (Line 1). In the constructor (Lines 7-12), we initialize `Queue` objects, which keep the actual packets. The methods, `enqueue()` and `dequeue()`, are event handlers for packet handling. The `VIFlet` class has these methods and PriQVIFlet is overriding them. The JVM, embedded in a VIF, in-

```
1  public class PriQVIFlet extends VIFlet {
2      private static final int NCLASS = 4;
3      private Queue[] queue;
4      private PacketClassifier pc =
5                  new SimplePacketClassifier();
6
7      PriQVIFlet() {
8          System.out.println("Initializing...");
9          queue = new Queue[NCLASS];
10         for (int i = 0; i < NCLASS; i++)
11             queue[i] = new Queue();
12     }
13
14     public void enqueue(Packet p) {
15         int type = pc.classify(p);
16         length++;
17         queue[type % NCLASS].enqueue(p);
18     }
19
20     public Packet dequeue() {
21         for (int i = 0; i < NCLASS; i++)
22             if (queue[i].isEmpty() != true) {
23                 length--;
24                 return queue[i].dequeue();
25             }
26         return null;
27     }
28 }
```

Figure 6.2: Sample VIFlet code

vokes `enqueue()` when a packet arrives, and `dequeue()` at an appropriate timing for packet drainage. Accordingly, the packet processing is done in a event-driven fashion. In the enqueue method (Lines 14-18), the VIFlet classifies incoming packets with a packet classifier, `SimplePacketClassifier` class (Line 15), and enqueues packets based on the flow class returned by the classifier (Line 17). The dequeue method (Lines 20-27) looks for a non-empty queue, and dequeue a packet (Line 24).

Although the VIFlet model is quite simple, the example suggests that the model has enough descriptive power to realize queuing controls. It is also easy to see that it may serve as a packet filter, by selectively dropping packets in the enqueue function.

VI.1.2 Execution environment - a lightweight in-kernel JVM/JIT

We prototyped an execution environment for VIFlets, as an extension to the hierarchical virtual network interface framework. On top of the virtualization framework, an in-kernel lightweight JVM and a Just-In-Time compiler are embedded, which are explained below.

VI.1.2.1 NVM - A lightweight Java Virtual Machine

The Netnice Virtual Machine (NVM) is a lightweight Java Virtual Machine, based on Waba VM [137]. Waba is an open source Java interpreter, originally designed as a portable JVM for mobile devices. However, it requires an independent class library file and the class loader cannot take a Java archiver (JAR) file as input. Besides, for easy integration with the kernel, the virtual machine needs to be more compact than the current Waba.

For this reason, we implemented a mechanism to embed the class library into the JVM (ROMizer), and a decoder module of Java archiver. Additionally, we thoroughly customized the VM core and the standard class library for VIFlet execution.

The resultant VM is kept just about 64KB in its size, including the class library. To keep the binary size compact, the built-in class library is heavily tuned for VIFlet execution and not all of the standard Java classes, such as `java.Math.*`, are supported. Nevertheless, the VM core can execute many ordinary Java class files and JAR-archived class files generated by ordinary Java development environments. Note that the size of the Sun classic JVM for the same architecture is 981KB, and it requires a library file of 8907KB (JDK1.1.8).

The user interface is implemented utilizing the same file system abstraction of the VIF system; users write a VIFlet class or a VIFlet archive file onto `viflet` file in a VIF directory, to inject the code. There are two additional files in the directory, `log` and `conf`, which are used for logging of the execution and for configuration of the VIFlet (see Figure 6.1).

VI.1.2.2 NYA - A Just-In-Time compiler

Although NVM can execute VIF files as well as ordinary class files, interpretation is not a practical solution to execute programs in the kernel. A natural solution here is to utilize a Just-In-Time (JIT) compiler. The main challenge is creating optimization methods specifically for boosting of packet processing code.

Accordingly, we developed a new JIT for NVM, reusing code generation framework of TYA [78], an open source JIT for Sun Classic VM on IA32 architecture. Because of the difference of the memory model, memory layout of the Java stack and the heap are totally different, and they differ in how classes are represented on memory. Consequently, almost all code was rewritten and we added various optimization algorithms to boost the execution performance. The optimization options available on the system are shown in Table 6.1.

OPT_REGCACHE	Register cache of locals
OPT_COMBINEOP	Instruction folding
OPT_COMBINELD	Further instruction folding
OPT_ACACHE	Enables array object cache
OPT_OBJECT	Enables fast object access
OPT_INLINING	Inlines small functions
OPT_NULLIFY	Nullifies void functions
OPT_INLINECK	Inlines clock syscall
OPT_FASTINV	Invokes methods fast
OPT_MCACHE	Enables method cache

Table 6.1: Optimization Options

As shown later, the simple table-driven JIT for NVM, called NYA, boosts execution of programs by a factor of six to eleven, compared to the interpreter execution. Like NVM itself, NYA is also just about 64KB in its size. To the best of our knowledge, NVM/NYA is the smallest Java execution environment that can achieve performance of this degree on IA32.

VI.1.3 Lessons learned

To close this section, several lessons leaned in the implementation process are summarized, as they might benefit future efforts in the virtualization of network I/O.

VI.1.3.1 Coding process Development process of in-kernel virtual machine has rarely been reported in publication. Accordingly, we briefly summarize the coding process and make comments on the tools we developed to facilitate the coding process (Figure 6.3).

First, before developing the virtual machine, we needed to develop the VIFlet programs to test the functionality of the virtual machine, and the class library which supports the execution. For example, a VIFlet needs `Packet` class and `Queue` class, which must be provided in a class library. Accordingly, in the initial phase of the development, we concurrently programmed sample VIFlets and the class library. To facilitate the coding process, we implemented a Java tool, `viflettester`, which tests basic functionality of target VIFlets and the class library on IBM's Eclipse. The tool comes with a built-in traffic generator and a driver to emulate packets out of a dump file of Berkley Packet Filter [88], so that we can

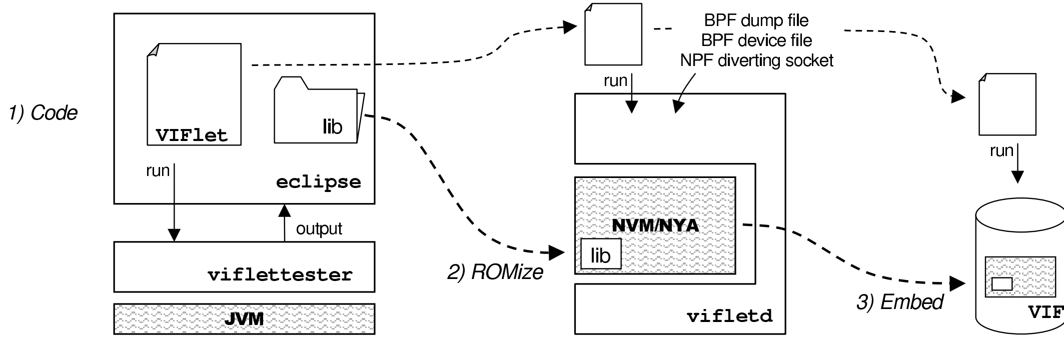


Figure 6.3: Overview of the coding process

test the programs in various manner.

Second, we implemented the virtual machine (NVM) and its JIT compiler (NYA). To simplify the debugging of the lightweight JVM/JIT, we first coded and debugged the VM at userland. For this purpose, a daemon, `vifletd`, was implemented, as a driver of the VM core being built. The driver takes two inputs, a VIFlet and packet data. There are three packet data sources: a dump file of BPF, a device file of BPF, and a diverting socket of Netnice Packet Filter [101]. Although the BPF mechanism is used only to inject packets to the VIFlet, packet diverting feature of NPF realized actual insertion of VIFlets onto the actual packet data path, executing a VIFlet as an application at userland.

Another issue was the integration of the class library and the virtual machine. For this purpose, we developed a program, ROMizer, which converts the class library into a form which can be merged onto the VM source. This mechanism facilitates easy extension of the VIFlet functionality, by allowing easy modification of the class library embedded in the virtual machine.

Lastly, the developed virtual machine is integrated into the VIF framework in the kernel. The process is to embed appropriate hooks in the VIF functions so that packets are diverted to the VM core in the kernel.

VI.1.3.2 Porting JVM into kernel space There are many standard functions, which are commonly used for ordinary programs, but not supported in the kernel. Obvious examples include I/O functions like `read()` and `write()`, which are used for class loaders on

virtual machines. These functions require special consideration for the kernel porting. Likewise, there are many functions which have different argument set in the kernel, such as `malloc()` and `free()`. These functions can be well substituted by short macros, or wrapper functions. There are another set of functions, which were thought to be unsupported, but work well even inside the kernel. For example, abort processing is frequently used in a VM code, and is implemented with `setjmp()` and `longjmp()`, which worked well even inside the kernel.

Just-In-Time compilers sometimes utilize hardware protection mechanisms, such as "division by zero" and "page fault", to simplify runtime checks. However, inside the kernel, it compromises modularity of the virtual machine, because this requires hooks to the VM code, in the hardware interrupt handler. Because such a hook highly complicates the system design, we substituted the runtime checks by inserting extra instructions into the code generated by the JIT.

Further, to avoid huge cost for one-shot compilation at code injection, we implemented a delayed (lazy) compilation mechanism. The mechanism defers compilation of methods until they are actually invoked. In this scheme, a code invoker function checks if the method to run has a compiled binary, and if not, it dynamically calls the JIT to translate the bytecode into a native binary. The binary will be simply executed afterward, amortizing the compilation cost. Note that the hot-spot detection is less important here, because packet processing code is on the critical path by nature.

VI.1.3.3 Customization for performance boosting To efficiently execute VIFlet programs inside the kernel, further customizations are made on the VM. Because most variables for packet processing do not require 64 bits, 64-bit variables such as `long` and `double` are substituted by 32-bit `int` and `float`. This change greatly simplifies the VM design on a 32-bit architecture. Another enhancement was accomplished by limiting several features, such as `Exception`, `Thread`, and `Monitor`. The VM simply aborts when it finds such operations, as a security violation. Users are expected to assure the code integrity, avoiding the use of such features.

Accordingly, NVM supports a subset of the VM features, not the genuine Java specifica-

tion. This approach looks common for specialized Java execution environment. For example, Java 2 Micro Edition (J2ME) is specifying such a VM, the K Virtual Machine, designed for embedded systems with limited memory and processing power [123]. We took the same approach for packet processing. Note that, despite the changes, NVM does support standard Java mechanisms such as inheritance and interface and can run ordinary Java bytecode (such as HelloWorld.class) in the kernel, without any modification.

VI.1.3.4 Reducing cost for runtime checking It is important to optimize data access to the packet payload, which needs costly runtime check of address legitimacy. In particular, it would be prohibitive to calculate checksum if we check the addresses in each step of the payload access. Accordingly, we provided native functions for routine processing of this sort, including checksumming. It is also a reasonable choice to provide a standard packet classifier as a native routine, not to harm system performance.

Another peculiarity is reference to packet objects in the heap. If a packet is no more referenced, we want to discard the packet as quickly as possible, not to exhaust packet buffer pool on the system. However, this checking can be expensive. Suppose that a careless programmer enqueues a given packet into a queue, and then, put the same packet into another queue. This operation results in duplicated references from multiple queues, and the system consistency can be compromised. The straightforward solution is to sweep the heap, not to leave a bogus reference, when the packet is first dequeued. But, this is obviously impractical. Accordingly, we propose not to allow such a duplication of packet reference in a VIFlet. Such a check is made at runtime relatively easily, by maintaining a reference counter on each packet object. The system simply aborts the execution when duplication is detected. This is another form of optimization for packet processing code, and we believe that this is not a serious restriction in packet processing programming.

VI.1.3.5 Garbage collection and protection mechanism Another concern to run a JVM in kernel is the garbage collector (GC), because execution of GC in the kernel may freeze the system for a certain period of time, severely impacting system performance. However, detailed code review suggested that actual event handlers rarely allocate memory dynami-

cally, because of the overhead. For this reason, we decided to make it the user's responsibility to write programs which preallocate necessary memory in initialization phase and do not use `new` in the handlers, not to start garbage collection on-line. We believe this is not a severe constraint for packet processing code. Note that, if `new` operation is not called, GC never starts.

However, it is possible that a malicious user may run a program with a thousands of `new` instructions, or a careless programmer may leave an infinite loop in his VIFlet. Accordingly, we implemented two protection mechanisms: a stack depth counter which aborts the execution when the stack depth exceeds certain threshold and a software timer mechanism which aborts VIFlet execution when a time threshold is exceeded, for detection of infinite loops.

VI.2 QUANTITATIVE EVALUATION

This section profiles the prototype implementation. First, the lightweight in-kernel JVM/JIT is profiled. Then, overall performance of the VIFlet system is evaluated.

For this purpose, a couple of machines which have the following specification are used: Intel Celeron 2.0Ghz CPU with 512MB main memory and a Gigabyte GA-8INXP motherboard (Chipset: E7205). The motherboard has an on-board GbE NIC, the Intel PRO/1000, and we added another to the system, the Intel 82559 Pro/100 Ethernet. We connected the GbE interfaces of the machines with a GbE switch, a D-Link DGS-1008D, for the data plane, and the 100/10 interfaces with another switch, a D-Link DI-604, for the control plane.

For their operating systems, we used FreeBSD 4.11 system and a customized kernel which supports the VIF and the NVM system. We also installed JDK 1.1.8 for compilation of Java programs and for the comparative studies described in the section.

VI.2.1 NVM/NYA Module Performance

VI.2.1.1 Overall Performance First, we performed the Sieve benchmark, widely available for performance evaluation of Java platform. It is a simple benchmark that iteratively

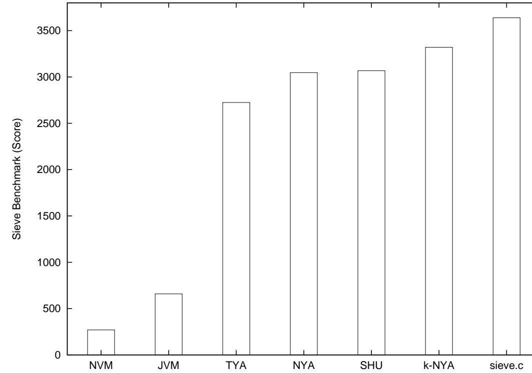


Figure 6.4: Sieve Benchmark

performs the Sieve of Eratosthenes algorithm, and shows the number of iterations in a period, as Sieve score. We used this benchmark to compare our implementation with other JVMs and JITs. In this section, all the benchmarks are executed at userland, unless otherwise notified.

The result is shown in Figure 6.4. NVM denotes our interpreter performance, which had the lowest performance. Next to NVM is the Sun classic interpreted JVM, bundled with JDK1.1.8, which performed 2.4 times faster. TYA and SHU (shuJIT) are open source JITs for the Sun JVM. As illustrated, these JITs boosted the execution of the algorithm by a factor of 5. Our JIT, NYA, was equal to shuJIT, exhibiting substantial improvement over the NVM interpreter. Performance of the JIT inside the kernel (k-NYA) was also measured. In the experiments, the CPU protection feature described in the last section were turned off, to execute the benchmark which surely exceeds the CPU cycle limit. Note that this setting for k-NYA would block the entire system. As shown, a slight increase in the score was observed, achieving the best performance among all the open source JITs. It is probable that the non-preemptive property of the kernel execution contributed to the performance gain.

In addition, the performance of a native program, sieve.c, was also measured, which performs the same Sieve algorithm. The JIT approach was 11 times faster than the interpreter, and 91% of the native performance; this implies that virtualized code does not jeopardize the system performance. Comparison with optimized code was not made because operating

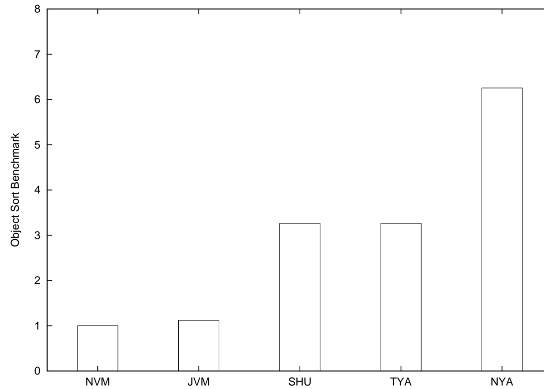


Figure 6.5: Object Sort Benchmark

system kernels are not always optimized. This is because OS kernels are located quite close to the hardware, and thus, optimizations may harm instruction ordering which might have critical importance to the code.

Secondly, the Object sort benchmark was performed, which is a simple benchmark to check performance of object handling and array access, by creating, sorting, checking object arrays. In this study, no counterpart was provided for the native code, because it is hard to make a fair comparison, compared to the case of Sieve benchmark in which we can run almost identical programs on the platforms.

The result is shown in Figure 6.5, as a relative performance compared to the NVM which had the lowest performance. As shown, shuJIT and TYA performed three times better than the interpreter, and NYA outperformed the other open source JITs. Compared to Sieve benchmark, which exhibited 11 times better, the benchmark did not show the great advantage, though 600% boosting is still a substantial increase.

VI.2.1.2 Performance Breakdown To further investigate the differences among the implementations and cause of the difference across benchmarks, performance breakdown of the implementations are measured next, by a Java benchmark suite. In this study, execution time to perform primitive operations of Java bytecode, such as method invocation, array access, ALU, object creation, etc, are measured. Because it is less informative here, and for

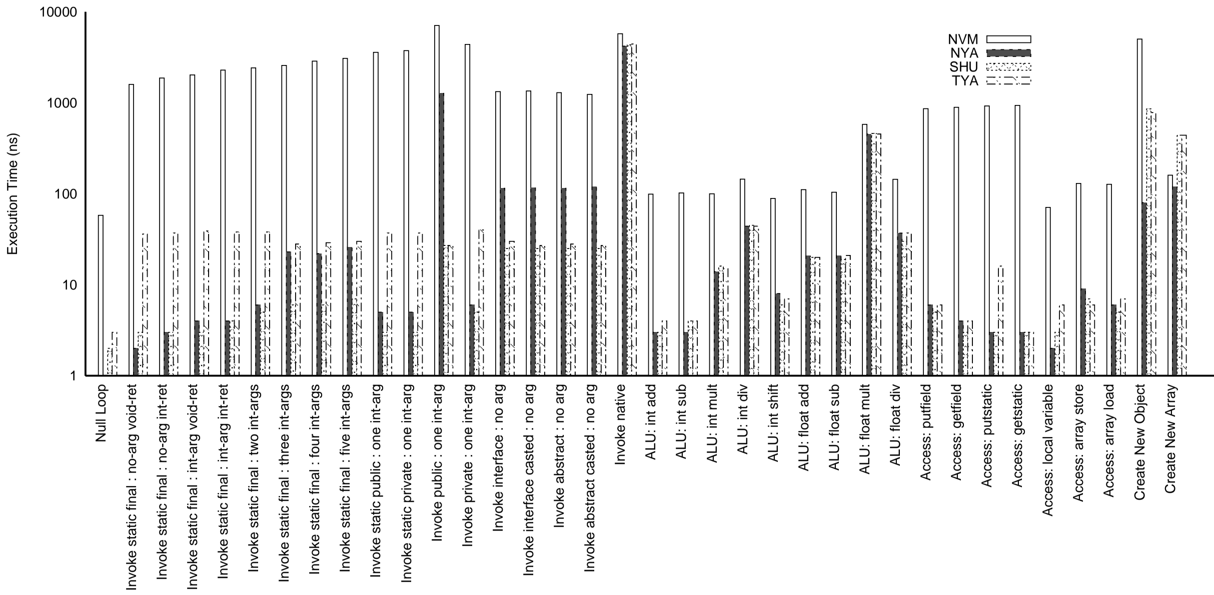


Figure 6.6: Benchmark of primitive Java bytecode operations

readability of the graph, performance of the interpreted Sun's JVM is not included in the figure.

The results are shown in Figure 6.6. At a glance, we notice that the JITs greatly improve performance of the interpreter (note that Y-axis is a log scale). For example, NYA makes execution of a null loop 60 times faster than interpreted NVM. For method invocations, it achieved up to 1000 times better performance. Meanwhile, in the invocation benchmarks, shuJIT has the best performance. This is because it has a weaker inlining criteria, and inlines the invoked methods more often than the other two, resulting in the better performance. "Invoke Native" function calls took almost the same for all the JITs. Note that the figures include execution time of the callee method, and could be inaccurate.

The JIT implementations did not show great difference in the ALU performance. "Access" operations (read and write on class fields and object fields) exhibited an irregular pattern; shuJIT again outperformed NYA and TYA, but static field access were even. A marked difference appeared in the access of local variables; NYA did the best. Interestingly, we observed an odd result in the performance of array access; NYA was not the best, contradicting to the result of Object sort benchmark. On the other hand, it created new objects

Class	Method	#insn	time (μsec)	time / insn
HelloWorld	main	4	117.8	29.5
HelloWorld	$\langle init \rangle$	3	35.5	11.8
Sieve	main	14	139.0	9.9
Sieve	runSieve	109	191.0	1.8
Sieve	$\langle init \rangle$	3	11.1	3.7
PriQVIFlet	$\langle init \rangle$	29	131.7	4.5
PriQVIFlet	enqueue	26	52.3	2.0
PriQVIFlet	dequeue	28	79.4	2.8

Table 6.2: Compilation cost sample

and arrays at the lowest cost.

These results suggests that we may need to i) reduce method invocation cost by carefully designing in-lining criteria, ii) reduce the native call cost, and iii) avoid packet object creation in the kernel. Because packet processing code is a hot-spot of network systems, it is well justified to preallocate necessary objects at system boot time.

VI.2.1.3 Compilation cost Lastly, the cost of dynamic compilation in NYA is studied. To sum up, overall compilation time took about 150 μsec for HelloWorld, 350 μsec for Sieve benchmark, and 260 μsec for the priority queue VIFlet (Table 6.2). Because kernel blocking of this amount could be unacceptable for performance-conscious systems, we investigated the cost in a little more detail below.

As shown in the table, the compilation cost per bytecode instruction greatly varies from method to method, and the cost is not strictly proportional to the size of the code. For example, in PriQVIFlet case, the compilation of "$\langle init \rangle$" is more costly than the others, though the instruction counts do not differ very much (26-29). It is also easy to realize that the first method for each class always marks the highest time-per-instruction (29.5, 9.9, and 4.5). This data suggests that there are other determinants of the performance, besides the instruction count, and we found that the compilation of the first method intensively calls the class loader to load basic classes from the ROMized class file image, such as java.String, java.StringBuffer, and java.System, inflating the initial cost.

Second, we studied the cost of optimization options. Because certain processing like

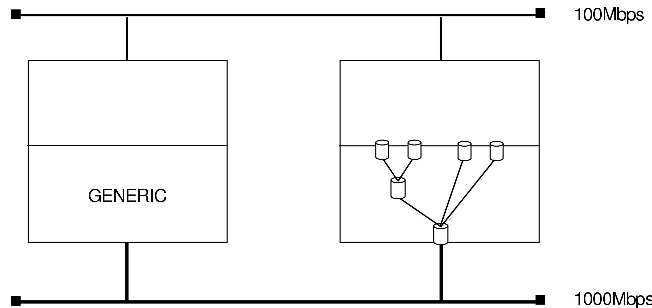


Figure 6.7: Experimental Setting

inlining clearly inflates compilation cost of a method, we expected that we can reduce the cost by turning off the options. However, we did not observe the expected gain. This was because compilation time of the first method dominated the overall cost, although significant improvements in cycle-per-instruction of the other methods were sometimes observed by tuning of the optimizations.

The results suggest that it would be reasonable to focus on the class loading cost of the first method, for further performance tuning. One approach is to modify the class loader so as to pre-load some basic classes (java.*) embedded in the VM at the system boot time. Because compilation cost for each method is still significant, the result supported the lazy compilation approach, avoiding huge cost for one-shot compilation at code injection.

VI.2.2 VIFlet system performance

Next, performance of the in-kernel JVM to run VIFlet programs is studied. The hardware setting used in the experiments are shown in Figure 6.7.

VI.2.2.1 A case study: priority queuing First, to demonstrate the system functionality, we coded a simple VIFlet, which gives priority to traffic with a certain TCP port. In this study, we generated background FTP traffic, while generating the target stream, both from the VIF host. At the receiving machine, we dumped all the traffic with `tcpdump`, and used the dump for further analysis. We limited bandwidth of the VIF to 20Mbps, to observe

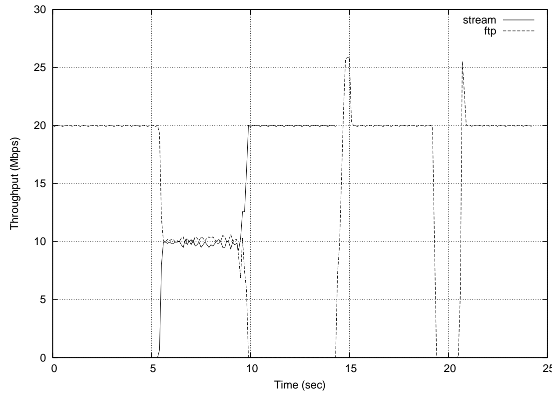


Figure 6.8: A case of priority queuing VIFlet

the priority queuing behavior of the flows.

Figure 6.8 shows the result. At time 0, the background FTP starts. At time 5, the target stream is started. At this point, no control is made yet, and thus, these two streams shared the available 20Mbps. Roughly at time 10, the priority queuing VIFlet is injected to the VIF. After a short adjustment phase, the target stream got all the 20Mbps, and the FTP traffic starved. This continued until the target stream stops roughly at time 14, followed by rapid recovery of the FTP traffic. The last event occurred around time 20, when the priority queuing VIFlet is removed. The recovery took longer this time, probably because several packets are discarded by the VIFlet removal and retransmission occurred.

As illustrated, the priority queuing VIFlet performed priority queuing, and controlled the packets as expected. It also demonstrated that a user can dynamically and effectively control his network I/O with virtualized packet processing code.

VI.2.2.2 A simple VIFlet performance To measure performance of the VIFlet system, we first measured throughput for a simple VIFlet, which just relays packets. For this purpose, we generated a TCP stream from one VIFlet host to another, measuring the throughput at the receiving program. The VIFlet machine was configured so that the measured traffic goes through just one VIF, attached to the GbE interface. We run the traffic generator for 10 seconds, and used the best 1 second interval for further analysis.

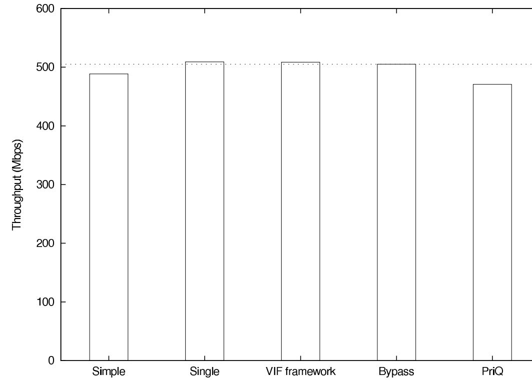


Figure 6.9: A simple VIFlet Performance

The result is shown in Figure 6.9. The simple VIFlet achieved 488.5 Mbps. The next bar is for a VIFlet which controls just outgoing traffic, reducing the processing cost approximately by half. "VIF framework" denotes throughput of the VIF system without running any VIFlet, and "Bypass" is the performance when we bypassed the VIF system, which serves as the baseline of the study. The last "PriQ" is for the priority queuing VIFlet, which performs priority queuing.

First, the result illustrated that the VIFlet framework works at a reasonable cost, just a few percent overhead of the max throughput. Second, in the case of "Single", we observed a slight performance increase by omitting processing of incoming packets. This is probably because latency for the processing of incoming packets (mostly acknowledgment packets) was bounding some TCP algorithm. Third, we observed almost no performance penalty in the VIF scheduling framework, suggesting efficiency of the implementation. In the case of priority queuing, the experiment exhibited a slight overhead, but still, it was about 7% of the peak performance, suggesting that a virtualized code can be a practical solution.

VI.2.3 Breakdown of a packet processing

Lastly, to study processing overhead of the virtualized code, detail of packet processing is studied. For this purpose, we inserted code fragments to sample system clock into time consuming portion of the packet processing path, mostly, into head of the first functions of

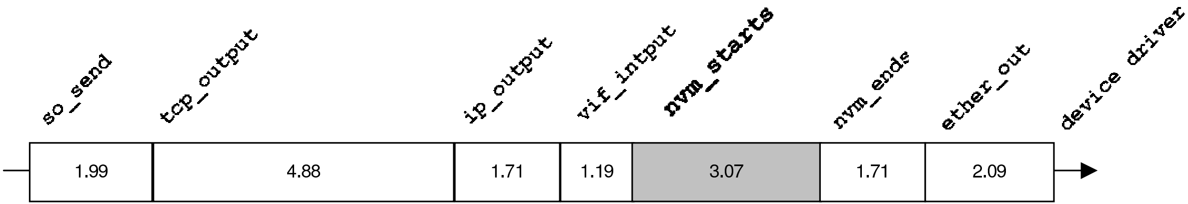


Figure 6.10: Breakdown of processing for an outgoing packet (in μsec)

each network layer. Then, we injected `SimpleVIFlet` into the GbE interface and generated an outgoing FTP traffic between the hosts, measuring the intervals between such checkpoints for a packet.

The result is shown in Figure 6.10. The figure indicates latencies between the checkpoints, starting from `so_send()` function, which is a socket interface, until the system calls the device driver function involved in the communication. All the numbers are in μsec. As shown, the first function, `so_send()`, took 1.99 μsec, which includes buffer allocation, and related processing. Note that data copy between userland and the kernel space is not included in the figure. Then, the function calls `tcp_output()`, which shares large portion in the processing (4.8 μsec). This is reasonable because TCP involves various algorithms and processing, more complex than other parts of the packet processing, which is a straightforward processing to send out a buffer into hardware by nature. Because of this reason, `ip_output` takes less time (1.71 μsec). The IP function then calls the VIF layer, which includes the virtual machine processing. The system first performs processing related to the VIF structure, and then, calls the NVM. The latency for the processing is, 1.19 μsec and 3.07 μsec respectively. The packet, which went through the VIFlet processing, is then passed to the Ethernet function, 1.71 μsec later, which takes 2.09 μsec to call the device driver.

The throughput for the FTP transfer, measured at the userland, was about 21MB/sec. This suggests that each packet is sent at 70 μsec interval, approximately. This includes overhead for the FTP application, disk access, and buffer copies. In the meantime, the VIFlet processing caused 5.97 μsec overhead, including the VIF framework. Although the overhead is greater than TCP (4.88 μsec), it shares just a limited portion (about 8%) in the

entire cost, and thus, the virtualization did not severely degraded the entire performance of the system.

VI.3 QUALITATIVE EVALUATION

Next, qualitative evaluation of the computational model and the implementation are made. First, achievements of the approach are reviewed, followed by investigation of its limitation. Lastly, special remarks on other in-kernel Java VM projects are made to clarify contribution of the work.

VI.3.1 Achievements

First, the contribution of the proposed model and the prototype implementation are examined. To this end, a brief summary of the related work is presented first, followed by discussions on computation models and implementations, respectively.

VI.3.1.1 Kernel extensibility for packet processing It is not a novel idea to execute packet processing software component in kernel. Modularity is inherent in protocol processing, and thus, it is a natural idea to encapsulate each process into software component. Accordingly, various efforts have been made thus far, which targeted extension of operating system functionality with encapsulated software components.

To the best of our knowledge, Stream [112] is the first work that proposed such a mechanism. Execution environments for efficient packet classification have been studied [91, 88, 13, 140, 19, 47]. Some researches extended functionality of the micro-machines for efficient processing of traffic statistics and for packet filtering inside the kernel [67]. U-Net/SLE (Safe Language Extensions) [134, 136, 135] extended the Java specification to securely execute user-supplied programs in kernel, and implemented a virtual machine on a device driver for efficient handling of packet data. ASHs [131] also proposed user-provided interrupt handlers.

In addition to the proposals for kernel extensibility, almost any operating system nowadays has a mechanism to extend its functionality through *loadable kernel modules*. Some operating systems support extension of network I/O functionality, utilizing the device driver API.

As there have been so many kernel extensibility mechanisms for packet processing, here arises a natural question: *why do we need another?* To clarify the needs for an alternative, and to justify the proposed scheme, advantages of the computational model are summarized first, followed by significance of the implementation.

VI.3.1.2 Computation model First of all, although kernel extensibility for network I/O is not a novel approach, it is never too redundant to emphasize the benefit of embedded Java technology in the programming of network I/O code. It realizes architecture-neutral execution of packet processing code, which greatly facilitates code deployment over the network. The simplified memory management, coupled with protection features, allows easy extension of kernel functionality, and encapsulation favors code reuse. Aside from the general advantages of Java based approach, there are three distinct contributions in the computation model, as follows.

1. Existing models provided either raw access to kernel internals without any abstraction (in the case of kernel modules, for example), or controlled access with overly specific abstraction (packet filter micro-machines). The VIFlet model provides *flexibility in the degree of abstraction and information hiding*, through modification of the class library and `VIFlet` class. For example, the system can provide `FullfledgedPacketClassifier` to experts and `SimplePacketClassifier` to novices, both of which are descendants of `PacketClassifier`. *Encapsulation* by the class abstraction accommodates various functionalities in a unified manner with flexible extensibility. For example, we may simply add class methods `Packet.create()` and `Packet.append()`, for packet generation and data encapsulation in the module. As illustrated, the programming model facilitates future extension of the VIFlet functionality. These properties contribute to the flexibility of the scheme, which provides great freedom to implement novel technology for expert hackers, while allowing easy extension of kernel functionality even for novice Java programmers,

with just one mechanism.

2. Existing models allow execution of provided code at fixed granularity, mostly per-network interface basis; the injected code is simply applied to all the packets that pass through the point of control, indiscriminately. The VIFlet model provides *flexible control granularity*, inheriting this unprecedented property from the hierarchical virtualization model. For example, a VIFlet can be inserted to a network interface, to a process network I/O, to a set of flows, or to a socket, etc, which has never been realized.

3. Existing models permit execution of packet processing code only by system administrators. In contrast to the traditional kernel extensibility schemes, the VIFlet model *allows every user and application to inject their customized packet processing code*. This has become possible by the hierarchical virtualization framework, which realizes *partitioning of resource consumption* and *access control* over it. For more complete safety, the model can be coupled with runtime checking mechanism. For example, detection of infinite loops can be made by the virtual machine, which may also perform checking of access rights to non-private objects, such as a system-wide parameter.

VI.3.1.3 In-kernel Java VM for packet processing In addition to the contributions of the computational model, the in-kernel Java VM implementation for packet processing also has remarkable contributions, which are summarized as follows.

1. This work produced an empirical proof that a virtualized code could be comparable to hardcoded code in its performance; even the most straightforward approach to run Java bytecode inside the kernel exhibited moderate performance. This suggests that more sophisticated approaches can make the virtualized code run still faster. Note that we never intend to claim that Java bytecode is the best solution. The contribution of the work is that it proved that Java bytecode can exhibit this this level of performance, at least. Further, the performance profile serves as a guideline for further optimization efforts in this domain.

2. There has not been an open source JVM which is so compact and so fast, to the best of our knowledge. Thanks to its modularity of the design, the lightweight JVM/JIT can be

reused for many other purposes. For example, it can be applied to virtualize other OS components.

3. Virtualizations of network I/O and system performance have mostly been pursued independently. The proposed scheme integrated them, through dynamic optimization of packet processing code. The prototype suggested that virtualization improves flexibility, code portability, and even performance in network I/O processing. An exception is the Application Layered Framing approach [129, 45] which aimed at performance boosting with virtualized network interfaces. However, as justified in Chapter II, kernel approach is more suitable for network I/O, from a resource management point of view.

VI.3.2 Limitations

The limitations of the proposed scheme are examined below, with respect to the computational model, protection, and performance.

VI.3.2.1 Computational model The computational model, VIFlet, allows users to flexibly extend functionality of the basic `VIFlet` class, by inheriting it and overriding event handler methods. This encapsulates packet processing into software components and users can inject code into their VIFs, to realize a variety of controls at flexible granularity. Thanks to the hierarchical organization of virtualized interfaces, the executions are isolated and do not interfere each other.

Although this model looks promising, the proposed VIFlet model is still incomplete in several aspects. An illustration is a protocol processing VIFlet, such as IP VIFlet and TCP VIFlet. It is clear that IP VIFlet must be called before we call a TCP VIFlet for incoming packets. Likewise, IP VIFlet must be called just once in the packet processing path. Accordingly, the VIF framework needs to impose some rules about the ordering of the VIFlets in the VIF tree, which clearly contradicts to the assumption that users are allowed to execute any VIFlet on their VIFs. For this reason, packets are kept read-only in the current VIFlet implementation, which is a severe limitation as a computational model.

Another illustration is advanced queuing controls. As exemplified, the VIFlet model

realizes priority queuing, which processes packets whenever a link is available. This is called work-conserving scheduling. However, there is a class of scheduling, non work-conserving scheduling, which behaves differently. A typical case is Class Based Queuing [57], which regulates scheduling of packets based on monitored link usage. Such control requires a timer mechanism, to control timing of packet dequeuing, which is not yet supported in the model.

As illustrated, the VIFlet model still has several limitations. First, the computational model does not allow modification of packet payload. Second, the model does not allow creation of packets, because packets are kept read-only in the current implementation. Third, the model does not support timer mechanism. Accordingly, to replace the entire network I/O sub-system of modern operating systems, the VIFlet model still needs to be extended, for example, by adding methods to classes, such as `Packet.append(byte[])` and `System.timer(int)`. Implications of these changes are further discussed in the next chapter.

VI.3.2.2 Protection For safe execution of user-supplied code inside kernels, a variety of approaches have been already proposed, such as Proof Carrying Code [95], safe languages [56], software fault isolation [130], and protection from resource hoarding [116]. The VIFlet model also provides such protection, through isolation of the VIFlet execution. Because a VIFlet is authorized to access only to the packets delivered to the particular VIF, the VIFlet model can safely execute even network-born packet processing code, which was explored in [104, 105]. This is another form of sandboxing of code execution, reported in recent studies [36, 121, 61, 3].

One concern for the extensibility is how to verify the correctness of the program. Actually, the prototype does not perform code verification before execution. The execution environment guarantees system safety by aborting VIFlet execution when an illegal instruction is executed or when the CPU time limit is exceed, as described in the implementation section. Although the ostrich approach is simple, it is not always the best solution. Accordingly, it is still an open issue whether the current design is justified for its simplicity, or we may need to have a bytecode verifier also in the kernel.

Another concern is further protection and control of CPU cycles. In the current implementation, VIFlets request packet drainage by setting a flag (queue length variable), shared

with the VIF scheduler. Accordingly, although the system guarantees termination of method execution by the timer, a VIFlet may easily waste the CPU cycles by repeatedly requesting CPU. To prevent this, we would need to implement another counter mechanism, to prohibit such a "cheat".

Accordingly, there are several items left as future work about the protection. First, we need an appropriate justification not to have a verifier in the kernel. Conversely, it might be well justified to combine the kernel VM with a trusted verifier at the userland. Second, CPU scheduling and resource protection might need more detailed investigation. In particular, the trade-off between system safety and performance overhead needs to be addressed. It is also a reasonable choice to investigate more flexible CPU scheduling, by integrating it with task scheduler of the host.

VI.3.2.3 Performance The prototype implemented a lightweight JVM, which executes Java bytecode. Because interpretation significantly harms system performance, a Just-In-Time compiler was implemented for boosting of the performance. Nevertheless, results from the Sieve benchmark suggested that the generated binary performs worse than a non-optimized C code. This is mainly because there are wasted registers in the dynamically generated binary, whereas the non-optimized C code can make full use of the available registers on the processor. Although aggressive optimization of register usage is not performed in the implementation due to its translation overhead, sophisticated Just-In-Time compilers are dynamically optimizing register allocation nowadays. Accordingly, it might be reasonable to attempt such an optimization.

However, dynamic optimization of Java bytecode is costly, because the register reallocation needs data flow analysis of the compiled code. Rather, it would be more reasonable to develop an Instruction Set Architecture more suitable for dynamic code generation and dynamic optimization of running programs than Java bytecode, as explored in recent projects, such as LLVA [4], Dis [138], and Virtual Processor [106]. Further, although we hardcoded a customized packet classifier for efficiency in the study, it would be more reasonable to invest in a technology which automatically generates such a code, as is pursued in DPF [47]. Please note that the advantages of a computation model based on the Java *language* would be kept

valid by a cross-compiler, even if the Java *bytecode* is replaced by better ones.

There are other sources of performance gain in network I/O. One approach is to, rather than attempting to localize changes and minimize impact of virtualization, actually increase the percentage of the virtualized portion in entire packet processing, which we call *virtualization ratio*. This is because a higher virtualization ratio allows more aggressive customization of packet processing, as pursued in [129, 45]. Examples of such optimizations include Integrated Layer Processing [21, 20], which tried to fuse the layers, like the loop-fusion approach in compiler optimization, and a specialized protocol stack to benefit an application protocol [94]. In this regard, the profiling of packet processing shown in Figure 6.10 suggested that the virtualized code occupies just a limited portion of network processing. Accordingly, it is highly probable that, by extending the coverage of virtualized code execution to the entire network processing, we will have more optimization opportunities for further performance boosting.

Another approach is to increase the flexibility, in addition to the virtualization ratio, so that users can execute more powerful programs inside a VIF. For example, an in-kernel web server eliminates costly buffer copies between the kernel and the userland. Other applications, which are made possible by the flexibility, include speculative execution and customized caching, both of which greatly improve latency and performance of the system. In this regard, the read-only property of VIFlet is severely limiting applications of the VIFlet scheme, and thus, more flexible packet handling must be supported, such as modification of packet payload and generation of packets.

VI.3.3 Special remark on in-kernel Java VM projects

It is not a novel idea to use the Java programming language and a Java virtual machine for coding of packet handlers. For example, U-Net/SLE [136] proposed to embed a Java interpreter into the kernel so that users can inject original interrupt handlers. JEM [71] tried to integrate a Java execution environment and a microkernel, allowing execution of Java application in the network layer.

Hence, we have *absolutely no intention to claim originality in utilizing in-kernel Java*

for network I/O programming. The essential difference between the approaches lies in the control models. All the existing proposals work either on actual network interfaces, or in the main packet processing path. We proposed to virtualize the interfaces into a hierarchical structure, and demonstrated the great advantages of the model. First, although existing models are used only by system administrators, the VIFlet model allows every user and application to flexibly extend functionality of their VIFs. Second, although existing mechanisms can execute a program at just a fixed location, the VIFlet model realizes flexible control granularity.

We also differ greatly from the aforementioned projects with regard to the implementation. The virtual machine used in [136] did not even support multiple classes, let alone inheritance, forcing programmers to pack all the code into a single class. NVM has a JIT and supports all of these features, realizing better compliance to the language specification, though not to say perfect.

We admit that Ahead-of-Time (AOT) approaches have a performance advantage over interpreted or simple JIT based schemes. For example, Jaguar2 [132] used `gcj` [119] as an AOT compiler, circumventing the compliance problem, and even the performance disadvantage of Java bytecode. However, there have been better Instruction Set Architectures (ISAs) more suitable for dynamic optimization than Java bytecode or native binary [4, 138, 106]. Accordingly, it is doubtful whether AOT approaches will continue to maintain their advantages in the future, because dynamic optimization techniques are becoming increasingly popular in the research field.

VI.4 SUMMARY - "THIS IS PROMISING"

This chapter presented a framework for kernel extensibility of packet processing, the *VIFlet*. The model virtualizes packet processing code inside a VIF, and allows system users to flexibly extend their VIF functionality, by injecting their customized packet processing code written in Java. The embedded Java technology realizes architecture-neutral execution of packet processing code, which greatly facilitates development and deployment of packet processing

software.

The chapter first presented the design of the VIFlet scheme, followed by a description of a prototype implementation. As a code execution environment, a lightweight Java Virtual Machine was implemented and embedded onto each virtualized interface. To further boost the system performance, an in-kernel Just-In-Time compiler was also implemented.

Secondly, a systematic evaluation of the proposed model and the prototype implementation was made both quantitatively and qualitatively. The systematic evaluation exemplified the efficiency of the prototype implementation, comparable to hardcoded packet processing code, owing to the dynamic optimization of packet processing code. There has not been an open source Java VM so compact and so fast, not to mention being reusable for other projects. Qualitative evaluation illustrated significant contribution of the proposed scheme. First, existing mechanisms allow either raw access to kernel internals (kernel modules and device drivers) or limited access (packet filter micro-machines), whereas the VIFlet model covers a broad spectrum of possible abstractions in a unified manner. Second, existing mechanisms can insert modules into the kernel, but it is mostly onto a fixed location, whereas the VIFlet model realizes flexible control granularity. Third, existing models are used only by system administrators, whereas the VIFlet model allows every user and application to flexibly extend functionality of their VIFs.

Meanwhile, limitations of the model and the prototype were also identified. First, the read-only property and the lack of a timer mechanism in the proposed model restrict possible application and network control realized by the scheme. Second, the model simplifies the verification process. Although system safety is provided by an ostrich approach, it is still an open issue whether the current design is justified by its simplicity, or not. Third, a quantitative evaluation suggested that Java bytecode is not a desirable solution for dynamic code generation and optimization, critical for system performance. There is a need for better Instruction Set Architecture, more suitable for such purposes. Lastly, a qualitative evaluation suggested that the ratio of the virtualized code compared to the rest of packet processing, the *virtualization ratio*, needs to be increased, to further improve the performance.

VII TOWARD COMPLETE VIRTUALIZATION OF NETWORK I/O

The last chapter presented a computational model for network I/O programmability. For this purpose, a prototype was implemented on the hierarchical virtual network interface framework, and it was evaluated both quantitatively and qualitatively. In its evaluation, however, several limitations in the model and in the implementation were identified.

This chapter discusses these remaining problems, both in the virtualization model of network I/O and in the implementation, as a discussion chapter of the entire thesis. First, the virtualization model is discussed to clarify and break the limitations of the presented model. Second, implementation technologies are discussed toward further performance gain. Lastly, contribution and significance of the proposed approaches are examined by comparative studies with other noted approaches.

Through the systematic discussions, this chapter determines an ideal form of network I/O on end host operating systems, based on the materials presented in the dissertation.

VII.1 EXTENSION OF VIFLET MODEL FOR COMPLETE VIRTUALIZATION

The last chapter stated that packets are kept read-only in the current VIFlet model. Because this is limiting application of the computational model, this section studies how the virtualization model can be extended to support more flexible control. For this purpose, analysis of an actual protocol stack implementation is first presented, and then, complications of the VIFlet model are determined. These discussions are then summarized as a suggested design, which could replace the entire packet processing code of a modern operating system.

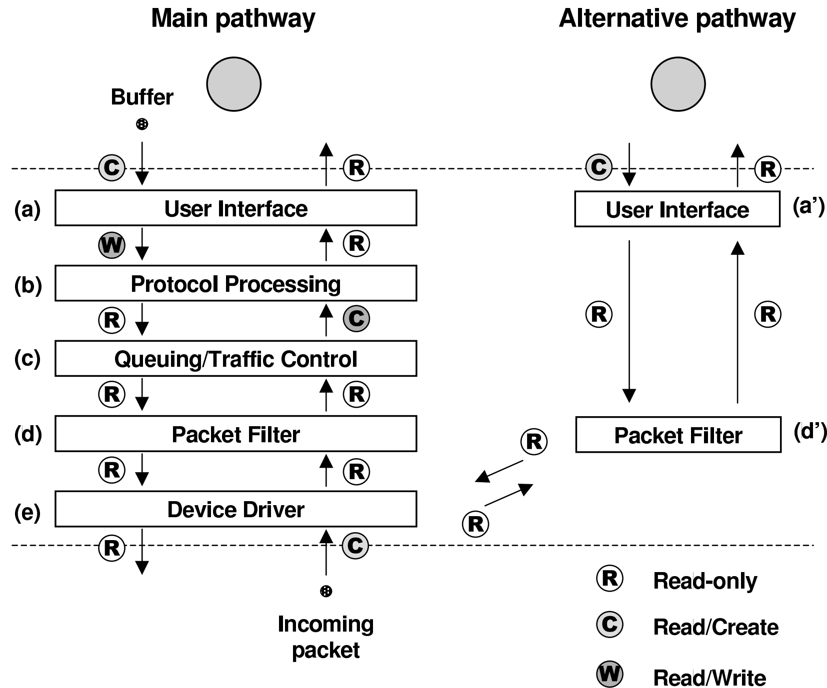


Figure 7.1: Packet processing locations and their properties

VII.1.1 Protocol stack implementation and Capabilities

The last chapter suggested several limitations of the VIFlet model. However, such limitations are not found on actual protocol stacks. Accordingly, the section first presents how packets are handled on actual systems, and each processing units are analyzed to compare the approaches.

Figure 7.1 illustrates conceptual breakdown of actual packet processing on a typical operating system. When a process at userland sends data, the buffer is first passed to a socket device, (a) User Interface. The interface creates packet data, and sends the data down to (b) Protocol Processing. The packet data is, then, processed by (c) Traffic control and (d) Packet filter modules, if employed, and passed to the (e) Device Driver for injection to the network. An incoming packet is processed in reverse order. The packet is first delivered to (e) Device Driver layer. The driver enqueues the packet to a queue for further processing. Depending on the configuration, (d) Packet Filter might be applied for security control, and (c) Traffic Control could be performed. Then, the packet is passed to (b) Protocol Processing code, followed by (a) User Interface for interaction with user applications. There is an alternative

pathway for packet monitoring applications. This pathway typically branches off (e) Device Driver, and (d') Packet Filter is applied to the flow, to specifically monitor packets of interest by a monitoring application through (a') User Interface.

As shown in the figure, each location (functional unit) has its own property, which is classified into three categories: *read-only*, *creates-packet*, and *modify-packet*. First, a module can be *read-only*, which is denoted as Ⓡ in the figure. In the discussion here, read-only indicates that the module does not modify packet data, nor create packet. For example, (d) Packet Filter classifies packets, accessing packet header or payload, but it never changes the packet data. Thus, it is called read-only. Similarly, a packet discarding operation is also considered read-only, because it never changes data. Please note that each unit in the figure has two entry points: for incoming packets and for outgoing packets. Because of this, each location has two properties. Second, a module can *create a packet*, denoted as Ⓒ in the figure. Packet duplication is also categorized here. This is obviously true for (a) User Interface and (e) Device Driver, where the system creates packet data structure for outgoing data and for incoming data, respectively. Processing (b) Protocol Processing also has this property for incoming data, because an incoming packet may initiate retransmission in TCP. Processing (a') also has the property for packet injection operation. Third, a module may *modify packet data structure*, which is denoted as Ⓦ. For example, (b) Protocol Processing may append header fields as a packet data goes down the protocol layers.

The diagram is useful to investigate the nature of packet processing on operating systems, and several insights are gained from a cursory glance, as follows. First, the limited virtualization, *filter* mechanism of the VIF framework, occupies locus (d) and (d'), which possess just read-only property. Second, the partial virtualization, the original VIFlet system, occupies locus (c), (d), and (d'), which also possess just read-only property. Because a timer mechanism does not modify nor create a packet, such a timer mechanism could be safely added to the VIFlet system, without causing complications.

Please note that there is an inclusive property between Ⓡ, Ⓒ, and Ⓦ; Ⓒ includes read access, and Ⓦ includes read access as well as packet creation capability. To understand packet creation capability of write access, recall that IP layer may fragment a packet, if the original data is too big, which generates several packets.

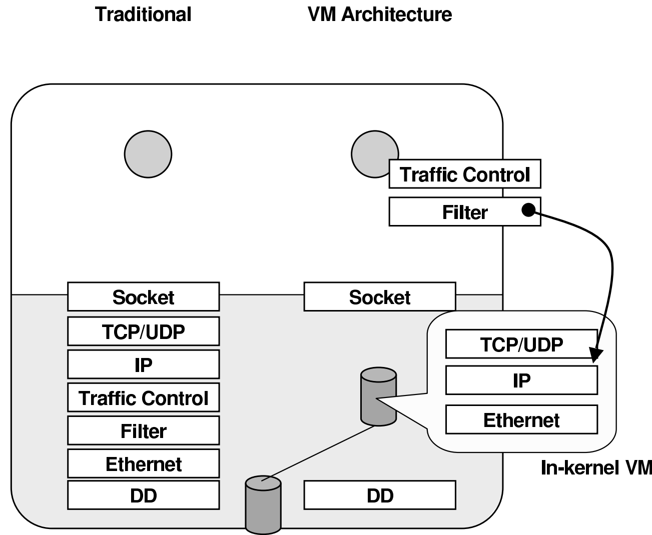


Figure 7.2: Imaginary overview of extended VIFlet model

VII.1.2 Capabilities and the VIFlet model

Now, to contrast the approaches, the traditional protocol stack implementation and the virtualization of network I/O code, a thought experiment is made by assuming that the VIF model supports protocol processing.

Figure 7.2 illustrates such an extended VIFlet model, which now supports protocol processing inside a VIF. The left side of the figure depicts the traditional architecture of the network subsystem, comprising layers of functions. The virtual machine architecture, on the right, replaces the protocol stack by a virtual machine with customizable packet processing codes. In such a framework, the protocol stack is virtualized and users are allowed to extend the stack, plugging in additional modules, such as packet filters, to meet their control needs. The device driver layer (DD) and socket layer (Socket) are kept hardcoded, because they are interfaces to applications and to hardware, which must be predefined.

The analysis of traditional protocol stack in the last section suggests that protocol processing requires Ⓒ creation and Ⓦ modification of packets, as well as read access to the packet payload. The following sections investigate complications of these capabilities on the imaginary VIFlet model.

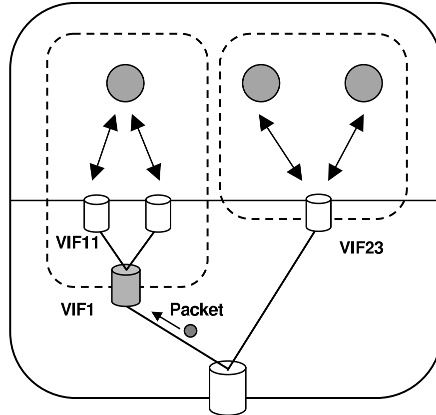


Figure 7.3: Write access and its complication

VII.1.2.1 Complication of Write access in the model There are three possible styles of write access to a packet buffer: modification of packet buffer, appending of data at the tail of the buffer (append), and insertion of data to the head of the buffer (prepend). Indeed, it is easy to extend the API to support these operations. For example, a simple method, `boolean Packet.write(int offset, byte value)`, would allow modification of packet payload. Another method, `boolean Packet.append(byte [] buffer)`, would be enough for the appending operation.

Although the extension of the API is simple, however, the change has serious implications. First complication is ordering of VIFlets, as suggested in the last section. For example, unless there is a rule that enforces ordering of processing, the system may apply IP layer processing to a packet, before applying TCP layer processing for outgoing packets. Likewise, unless there is a rule that prohibits redundant application of the same processing, the system may apply IP layer twice. This complication never happens to the original read-only VIFlet model.

Another complication is routing. Suppose that an incoming packet is routed to VIF11, and modification is made at VIF1 which changes the destination packet to VIF23 (Figure 7.3). If such change is permitted in the VIF tree, routing of a packet must be made each time a packet payload is modified. Even if the packet header is intact, checksum of the packet must be calculated again. Such a process is not easy because such a protocol could be performed by an upper layer protocol, such as TCP, or a lower layer protocol, such

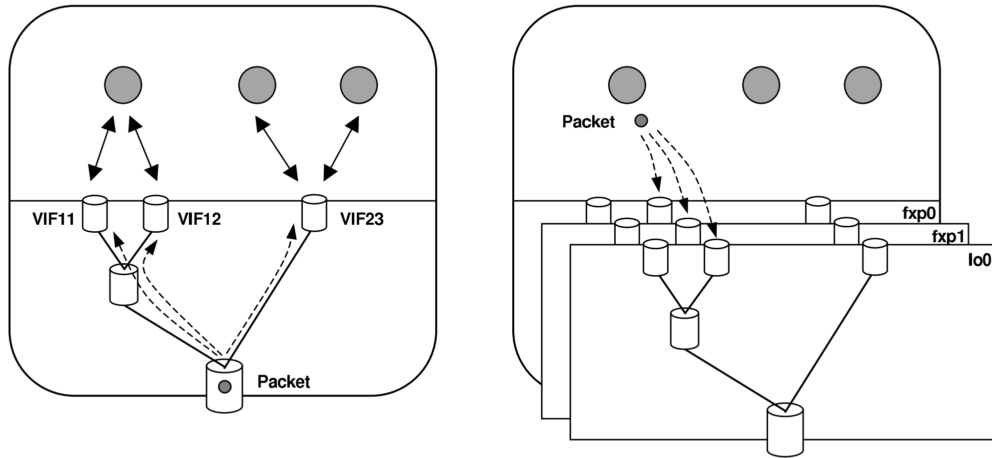


Figure 7.4: Creation capability and its complication

as IP. Accordingly, each VIFlet must possess all the protocol code, and this necessitates a mechanism to handle the VIFlet ordering problem, again. It is a possible option to ignore the change and to deliver the modified packet to its original destination. However, without recalculation of the checksum, the packet could be useless.

The complications suggest that the protocol processing requires a mechanism to manage VIFlets. Accordingly, protocol processing cannot be done just by a VIFlet. Let us define such problems that cannot be resolved just by a VIFlet, *externality* of the VIFlet model. Clearly, write access poses such a technical complication. The following section shows another form of the externality problem.

VII.1.2.2 Complication of Creation capability in the model Compared to the write access, packet creation capability looks more relaxed. For example, because packets are not modified in the VIF tree, it does not cause the VIFlet ordering problem. Likewise, it avoids the rerouting problem, because the destination is determined at the packet creation time. However, although packet creation is more relaxed than write access, it has other complications for both incoming traffic and for outgoing traffic.

For processing of incoming packets, a packet is first enqueued to a root VIF of a network interface (Figure 7.4-left). However, the system needs to determine a destination VIF of the packet, at the system entry point. Because there are multiple destinations, the system

must maintain a list of terminating sockets in the VIF tree. This necessitates a VIF-tree wide mechanism to keep such a list for incoming packets. Clearly, it is an externality of the VIFlet model.

Similar problem happens for creation of outgoing packets (Figure 7.4-right). If a host has multiple network interfaces, which holds in most cases because of the loopback interface, the decision to which VIF the packet is enqueued cannot be determined inside a VIF. Indeed, there is a need to keep a *system-wide* routing table, which is obviously an entity external to a VIF. Accordingly, outgoing packets also necessitate a system-wide mechanism. Clearly, it is an externality of the VIFlet model.

As illustrated, routing and demultiplexing operation require a system-wide mechanism. There exist other form of externality problems: how to switch network protocols (e.g. IPv4 and IPv6), and how to handle encrypted and/or fragmented packets which may complicate routing decisions. These problems cannot be resolved inside a VIF, and require a mechanism, external to the VIFlet model.

VII.1.2.3 Externality and the VIFlet model The preceding sections clarified that packet processing on the VIFlet model necessitates write access and packet creation, which cause a certain class of problems that are not properly addressed by each VIFlet. It follows that, to completely virtualize packet processing code on operating systems with the hierarchical virtualization model, we need to properly address the externality of the VIFlet model.

Utilizing the diagram in Figure 7.1, we can further clarify the externality of the model. Because routing and demultiplexing at (a) User Interface and (e) Device Driver are indispensable parts of protocol processing, it is not possible to virtualize only (b) Protocol Processing, but we will need to virtualize (a) and (e), as well. This implies that, when we replace (b) by a VIFlet, the VIFlet needs to modify (a) and (e). However, since (a) and (e) are shared by all the communications on the VIF tree, we cannot safely allow a VIFlet to modify them. Accordingly, the extended VIFlet model in Figure 7.2 cannot perform protocol processing.

It is possible not to use the VIF abstraction, as a framework. We might even allow flexible modification of the socket and the device driver layers, utilizing kernel modules. However,

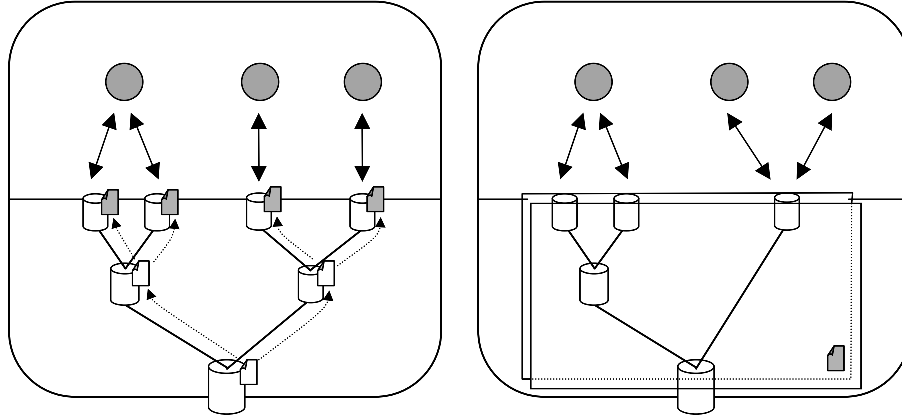


Figure 7.5: Two possible designs to assure consistency of protocol code

such approaches are problematic, as discussed in Chapter II. For example, without a resource management framework, users can easily jeopardize the system consistency. Further, such an approach spoils OS independence in packet processing code, which contradicts the original goal of virtualization. Accordingly, it would be well justified to take the VIF model for the code virtualization, rather than the traditional kernel extensibility approaches.

VII.1.3 Suggested Design

The above discussions suggested that, to completely virtualize network I/O processing, we need to address externality of the VIFlet model. First, we cannot simply virtualize the protocol processing part, and it must accompany virtualization of routing and demultiplexing. Second, this processing must be kept privileged for system consistency. Third, the code must be shared among all the communications. To meet these requirements, there are two contrastive approaches: overloading of the VIFlet abstraction and introduction of another abstraction.

The former approach is to use *inheritance* of properties, such as routing table and protocol processing code, to keep certain properties consistent throughout the VIF tree (Figure 7.5-left). In this approach, the shared properties are copied to each VIF, originating from the root VIF. In addition to their ordinary VIF processing, leaf VIFs performs routing service for outgoing packets, and the root VIF provides demultiplexing service for incoming packets. Protocol processing code is executed at the leaf VIFs, and other VIFs are kept read-only, to

avoid the complications of write access discussed above. The downside of the model is that it excessively overloads semantics on the VIF abstraction.

The other approach is to devise a software component outside the VIF tree, such as class `System`, which handles all the externality, such as routing and protocol processing (Figure 7.5-right). In this scheme, leaf VIFs call the software component for the packet routing service, and the root VIF requests the demultiplexing service. The VIFs are also kept read-only in this scheme, to avoid the ordering problem. The model does not overload the VIF abstraction, which greatly simplifies the system design. On the other hand, it requires an independent abstraction.

A concern is that, in either case, the protocol processing is kept privileged and users and applications are not allowed to flexibly execute their private protocol stacks. One solution to the problem is to assign a private network address to each user, or to each application, so that their network accesses are partitioned. This could justify the next generation internet protocol, IPv6, which holds a vast address space. Another concern is performance of the virtualized code. Because the routing and the demultiplexing are on the main packet processing path, efficiency of the execution would be critical for system performance, which is addressed in the next section.

VII.2 IMPLEMENTATION APPROACH FOR FURTHER PERFORMANCE GAIN

Next, the section discusses how to further improve performance of network I/O code.

VII.2.1 Research Trend

Researchers have been trying to improve performance of packet layers. The mbuf (Memory buffer) mechanism on BSD is such an attempt, which tried to eliminate buffer copies between protocol processing codes. Actually, many CPU cycles are spent for buffer copy between kernel and userland, and there have been many proposals to address the problem [32, 42]. Another source of performance overhead is the calculation of checksum. Consequently, it

is becoming popular to let the network interface controller calculate the costly checksum in high-performance systems [93]. These efforts have tried to reduce data cache pressure by avoiding data access by the processor.

In addition to the approaches that address the data cache misses, researchers have also tried to reduce packet processing cost by minimizing the instruction cache misses. This is because packet processing is not a good subject for the current processor architecture which hides the memory latency of instruction fetches, with instruction caching. For example, a protocol stack code does not include intensive loops, except for checksumming, and calls many functions in series, which greatly burdens the I-cache. Accordingly, to reduce the I-cache pressure, researches have tried intelligent I/O approaches in which special purpose processors designed for packet processing take over the burden of the processing [86]. Researchers have started to consider even an approach to integrate the CPU and the network controller on the same die, putting the transmission and receive buffers next to L2 cache [16, 18, 17]. Software approaches, such as Integrated Layer Processing [125, 33, 38, 52, 2, 87, 97, 23, 21, 25, 20] and code specializations [94, 39, 24] are also approaches to reduce the I-cache pressure.

As reviewed, main source of overhead in packet processing is the memory bottleneck. To address the problem, researchers have been addressing the problem for years, both in the D-cache and I-cache, from the software perspective and from the hardware perspective. However, these approaches investigated code optimizations in a static manner, and here lies a room for further optimization: The research community has been shifting from static optimization to dynamic optimization, utilizing runtime profile of ongoing programs [8, 43] and dynamic code generation [106].

These optimizations are advantageous over static optimization in that they can utilize information about the most recent branch biases and runtime constants, such as network addresses. Nevertheless, because packet processing code has been traditionally hardcoded inside operating system kernels, these recent technologies have been rarely applied to actual systems. Meanwhile, virtualization of packet processing code increases opportunities for such optimizations, as suggested by the last chapter. Accordingly, the following sections briefly discusses impact of the virtualization technology on several component technologies of packet processing, toward further performance improvement of network I/O systems.

VII.2.2 Components of packet processing and possible approaches

Below, components of packet processing are reviewed and possible approaches are discussed, to present a suggested design in the following section. The topics here include virtualization ratio, Instruction Set Architecture, language, compiler, profiling, and optimization.

VII.2.2.1 Virtualization Ratio Micro-benchmarking of the *filter* case illustrated that virtualized code significantly improved filtering performance. On the other hand, macro-benchmarking did not show significant differences, because protocol processing and buffer copy overhead dominate the entire packet processing cost. This observation indicates that even the best optimizer cannot improve system performance just by optimizing a limited portion of the entire processing, as Amdahl's law suggests.

Accordingly, the ratio of virtualized code in the entire processing is a decisive factor to improve system performance, and it is well justified to invest in the virtualization ratio. In this regard, an optimal strategy is to entirely virtualize the packet processing code, including the protocol stack. This is the ground for Application Layer Framing, which completely replaces the stack and provides the opportunity for aggressive specialization. Nevertheless, ALF requires raw access to the hardware, which causes resource management problems among competing applications. Although this is acceptable to dedicated systems, on general purpose operating systems, the kernel needs to provide proper partitioning and protection among resource usage by users.

The downside of such an OS service is its unavoidable overhead. Nevertheless, the flexibility of packet processing code it offers allows application specific processing, such as in-kernel caching. Actually, it realizes speculative execution, which significantly improves throughput and latency, even if the virtualization layer may cause unavoidable overhead.

The existing network I/O subsystem of operating systems has been hardcoded inside the kernels, and that has limited opportunities for such optimizations. By increasing the virtualization ratio, close to one, it would further improve system performance. In this attempt, it would be a reasonable option to allocate a network address to each socket, which simplifies packet demultiplexing and allows complete customization of protocol stack

associated with the socket. This could be another argument for IPv6, which holds a vast address space.

VII.2.2.2 Instruction Set Architecture The quantitative evaluation of the VIFlet system suggested that performance of the Java bytecode execution is limited by the translation cost and the poor register allocation. Consequently, to generate better quality of generated binaries, it would be preferable to have an instruction set architecture more suitable for dynamic code generation and for register optimization.

An approach for better register allocation is to use native binary, leaving optimization entirely to a static compiler. On the other hand, without any abstraction in the code, the context of data access is totally lost in native binary, and it limits further optimization at runtime. For example, we may reduce memory access by changing environment variables and host constants to an "immediate value" at load time, which is hardly possible once translated into binary. Besides, type checking of references would become harder, which compromises system safety.

Accordingly, the instruction set would be something similar to internal representation of compilers, which is more abstract than native binary, but is efficiently translated into the native code with optimized register usage. Such a strategy has been studied in Dis [138], Virtual Processor [106], vcode [48], and LLVA [4].

VII.2.2.3 Language and Compiler The discussion above necessitates an alternative of Java bytecode. But, does this spoil the Java programming language, also?

As a programming language for systems description, Java is a top-class solution. Descriptive power of Java for network system is exceeding. This is exemplified by many attempts which implemented even whole protocol stacks in Java [74, 80, 79]. As summarized in the last chapter, Java has a variety of advantages as a programming language for VIF extension. Advantages of Java are also substantial in its number of potential programmers and superb development environments. The language greatly simplifies the kernel coding task and would benefit many programmers who have even never thought of hacking the kernel.

Admittedly, there are several cases where a specialized language is more appropriate

and efficient. For example, DPF [47] dynamically generates packet classification code, more efficiently than a general purpose language, like Java. Lack of macro and C-style casting make protocol implementation in Java more laborious than C. Nevertheless, there are several ways to circumvent these limits.

Accordingly, it could be justified to utilize the Java programming language for description of the system, leaving exceptional cases for special purpose languages. To generate packet processing code in such an instruction set architecture that was discussed in the last section, a cross-compiler would be employed to bridge the gap between the Java language and the ISA.

VII.2.2.4 Profiling and Optimization The last component is profiling and optimization. Profiling technology basically optimizes binaries utilizing branch biases. Although this is also attempted by hardware branch predictions, code re-generation coupled with runtime profiling realizes better organization of code fragments and scheduling of instructions.

To realize such an optimization, there are two contrasting approaches: to perform the optimization in the kernel, or at the userland. In the former case, a kernel takes runtime profile, and optimizes the code. In the latter case, a kernel or a userland profiler takes the runtime profile, and a userland program optimizes the code, re-injecting it to the kernel later.

A factor that must be taken into consideration is cost of the code generation, as suggested in the last chapter. If the generation takes too long, it would be unpractical to perform it in-kernel because such operation freezes the entire system unless the kernel is preemptive. Accordingly, it is highly likely that in-kernel profiling and optimization would be computationally prohibitive. Meanwhile, even if on-line profiling is too costly, userland profiler can use packet dump files as input, to perform the profiling during system idle time.

Consequently, the userland approach would be more reasonable here, in that it makes effective use of system idle time and makes costly optimizations possible. What follows is a hybrid approach: a userland profiler optimizes the packet processing code in an ISA suitable for dynamic optimization at system idle time, and then it passes the optimized code into the kernel which again converts it to native binary after performing load time optimizations.

VII.2.3 Suggested Design

The review of component technologies suggested the following approach: A software component, like VIFlet, is first coded in the Java programming language. The code is then compiled into a target code, in an instruction set architecture suitable for dynamic optimization. The kernel takes such a code as input, and dynamically translates it to native binary, while applying simple load-time optimizations. A userland profiler monitors profiles the packet processing, performs code optimization at system idle time. The optimized code is injected again to the kernel, which translates it to native binary, replacing the old one. For system safety, we may also need a trust management mechanism, to protect the profiler.

Flexibility of network I/O code, realized by such an approach, is in great demand for performance improvement of network system. As reviewed in this dissertation, hardcoding of packet processing code is limiting performance of network I/O, and flexibility is needed for further optimization of packet processing software. Accordingly, virtualization of network I/O code will be increasingly important on end-host operating systems. Meanwhile, network I/O processing is being replaced by hardware, as observed in the popularization of checksum off-loading [93]. This trend would demand further flexibility in the networking I/O code so that we can easily replace certain processing by hardware. Accordingly, code virtualization is needed for future advancement of network I/O processing, both in software and hardware approaches.

Network I/O system on end-host operating systems has been hardcoded on traditional monolithic operating systems. This design does not have flexibility, and has been limiting performance of the network I/O system. The proposed scheme for virtualization of packet processing code would break such limit, by providing flexibility in the code. It would serve as a desirable framework for further advancement of network I/O on modern operating systems.

VII.3 CONTRIBUTION AND SIGNIFICANCE

To clarify contribution and significance of the virtualization scheme, the section compares it with other proposed virtualization schemes. For this purpose, a classification scheme of

	Quantity	Quality
Increase	(a) Virtual Machine	(c) Java bytecode
Decrease	(b) Grid	(d) raw access

Table 7.1: Classification of virtualization technology and examples

virtualization technology is first presented, to illustrate unique properties of the proposed virtualization scheme.

VII.3.1 Classification of virtualization technology

In system research, *virtualization* is a key concept, which forms the basis of the entire field of research. Many programs can be run on a machine without stepping over each other, because the operating system virtualizes the physical memory into partitioned domains, thereby protecting execution of each code from illegal memory accesses. It is also the virtualization concept that guarantees reliability of distributed systems, and realizes services with a capacity that far exceeds the capacities of individual processors and storage, virtually integrating the devices into a single entity. Virtualization, which has a long history, is not a novel idea. But still, it is a main theme of the research field, and is applicable to almost any aspect of computer systems and computing resources.

However, the term, virtualization, has several meanings. As just illustrated, it is used to multiplex resource usage for better utilization of a resource. Conversely, it has been used to create an illusion of an entity with larger capacity from individual entities with limited capacity. It is also used interchangeably with "abstraction". This being the case, this dissertation vaguely defined the term as *to modify a certain entity while keeping its predefined interfaces.* However, this is just a broad definition, and does not help in the analysis of the related technologies. We need a systematic classification system, rather than a definition, which provides an insight into the nature of technologies, to clarify significance of the hierarchical virtualization.

Table 7.1 is such a classification system. Class (a) is the most traditional virtualization technology, which is to increase the number of instances for better utilization of a shared resource. The Virtual Machine is a representative technology, and we are still observing advancement in this category, such as Xen [11] and VMware [128]. Class (b) is a technology to create an illusion of an entity, out of individual entities, for larger capacity and for better availability. Most distributed systems fall into this category, such as distributed operating systems and Grid Computing. These two classes of virtualization technologies affect quantitative property of virtualized object. Accordingly, these classes are labeled as such.

On the other hand, there are other class of virtualization technologies which do not affect quantitative property. These technologies are in the second column, labeled as “Quality”, simply because they do not affect quantity. An illustration of such virtualization is Java bytecode, in Class (c). It adds a virtualization layer between a program and a processor. As the case suggests, Class (c) of virtualization technology hides actual nature of the virtualized entity with an intervening layer which performs some operation, such as translation and auditing. Accordingly, it is categorized in the “Increase” row. Conversely, there is certain class of technology, in Class (d), that peels off such a layer, mostly for performance. An illustration is an API for raw device access, such as Virtual Interface Architecture [45], which provides standardized raw access to network interfaces for high performance I/O.

VII.3.2 Hierarchical virtualizations

This classification system looks complete in the sense that it covers any possible combination, because any virtualization technology either affects the number of virtualized entities or not. However, there are exceptional technologies which have properties of different classes. For example, a flexible virtualization scheme may increase the number of virtualized entity for better utilization and decrease the number for larger capacity, in a unified manner.

An example is web cache servers, which serve as a flexible front-end to back-end servers, balancing monitored server loads. In such a load-balanced server cluster, several caches may serve as proxies of one server (Figure 7.6-left), but they may also process requests for different

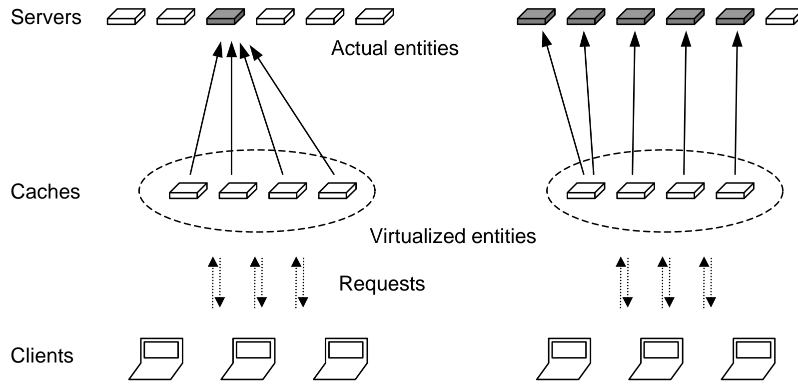


Figure 7.6: Flexible virtualization by cache servers

back-end servers (Figure 7.6-right). In the former case, the cache servers increase the number of virtualized entity, and in the latter case, they decrease the number of virtualized entity. Accordingly, such a caching mechanism may fit into Class (a) and Class (b).

Actually, *the hierarchical virtualization of network interface* is one such technology that does not properly fit into the categories. To understand the unusual properties of the hierarchical virtualization, let us examine a sample configuration of the VIFs in Figure 3.7. In this case, the root VIF is virtualized into three VIFs: VIF1, VIF2, and VIF3. Accordingly, we may say that the hierarchical virtualization is in Class (a) of the classification system in that it quantitatively increases the number of interfaces. However, as is realized by VIF31, the virtualization provides system protection at the same time; we protect interactive traffic of `sh`, by locking the `ftp` traffic in VIF31. Indeed, it affects the quantity by adding a paralleled instance, and affects the quality by adding an instance in a serialized manner.

This case suggests a unique property of hierarchical virtualization, in general, in that the hierarchical virtualization affects both quality and quantity. The following sections review several technologies which realized such a hierarchical organization of virtualized components, to further clarify the unique property of the hierarchical virtualization of network interfaces.

VII.3.2.1 Hierarchical CPU scheduler The first example is a *hierarchical CPU schedulers*, such as [62]. In a hierarchical scheduler, scheduling policy is represented in a hierarchical manner (Figure 7.7-left) and each scheduler accommodates schedulers which recursively

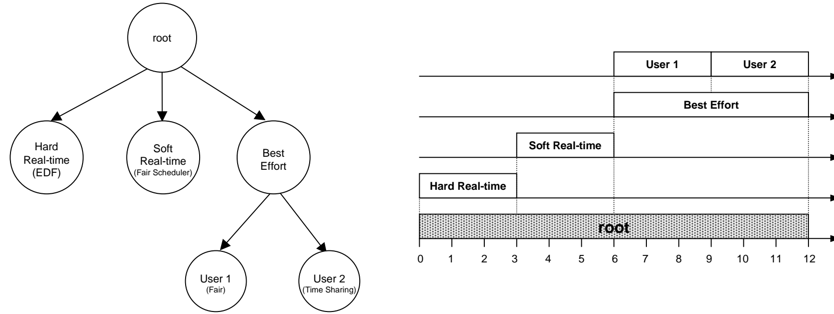


Figure 7.7: Hierarchical CPU scheduler

schedules tasks utilizing the time slot provided to it (Figure 7.7-right). Accordingly, hierarchical schedulers possess the *inclusive property*. In this case, CPU cycles provided for the root scheduler, is divided to task schedulers for Hard Real-time, Soft Real-time, and Best Effort. Then, a scheduler for Best Effort class schedules tasks for User 1 and User 2, utilizing the time slot given to Best Effort tasks.

On the other hand, this inclusive property does not hold in the hierarchical packet scheduler case (Figure 7.8). The left figure illustrates that there are two incoming packets in the hierarchical packet schedulers. The right diagram is a breakdown of its processing. As shown, Packet 1 initiates processing of a root VIF, and then, completion of the root VIF processing starts VIF1 processing, *sequentially*. It follows that there is no inclusive property in hierarchical packet schedulers, and hierarchical organization of VIFs is just defining hierarchical relationship between schedulers.

Actually, hierarchical packet scheduler and hierarchical CPU scheduler have *completely different characteristics and implementation*. Indeed, they resemble each other just in their appearances.

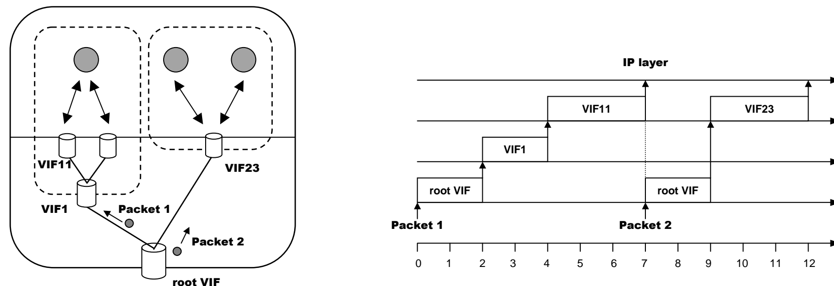


Figure 7.8: Hierarchical packet scheduler

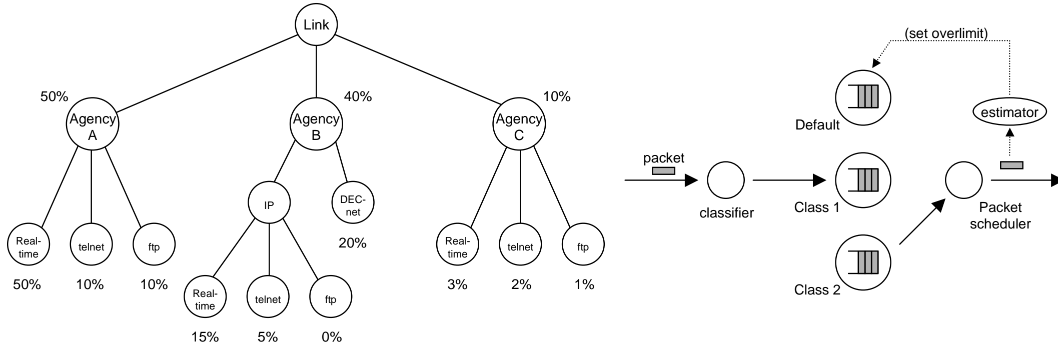


Figure 7.9: Class Based Queuing and hierarchical specification

VII.3.2.2 Hierarchical resource specification Next example is hierarchical specification of resource usage, as is used in traffic control models, such as Class Based Queuing [57]. CBQ is a traffic control discipline which allocates available bandwidth utilizing a tree-structured specification model, as shown in Figure 7.9-left. In this example, the link is shared by three classes, Agency A, Agency B, and Agency C, and each agency has its own resource use policy. Consequently, the specification appears in a hierarchical manner.

On the other hand, its actual operation does not work in a hierarchical way (Figure 7.9-right). As the diagram suggests, it maintains queues for specified classes in a flat manner, and the hierarchical organization is used just to determine resource allocation among classes.

This suggests that hierarchical packet scheduler and the hierarchical resource specification model have *completely different characteristics and implementation*. Indeed, they resemble each other just in their appearances.

VII.3.2.3 Hierarchical modeling of protocol processing Another example, is hierarchical modeling of protocol processing, as is often used in formalization of protocol stack operation, such as x-Kernel [66]. Figure 7.10-left illustrates hierarchical organization of protocol processing components in the x-Kernel scheme. A rectangle denotes such component, and a circle represents a session, which is dynamically created by the persistent components. A dashed arrow shows data movement between components, starting from hardware at the bottom. At first glance, it resembles the hierarchical organization of the VIF model, in that it is a tree structure which originates the network interface at the bottom and terminates at sockets in userland on the top.

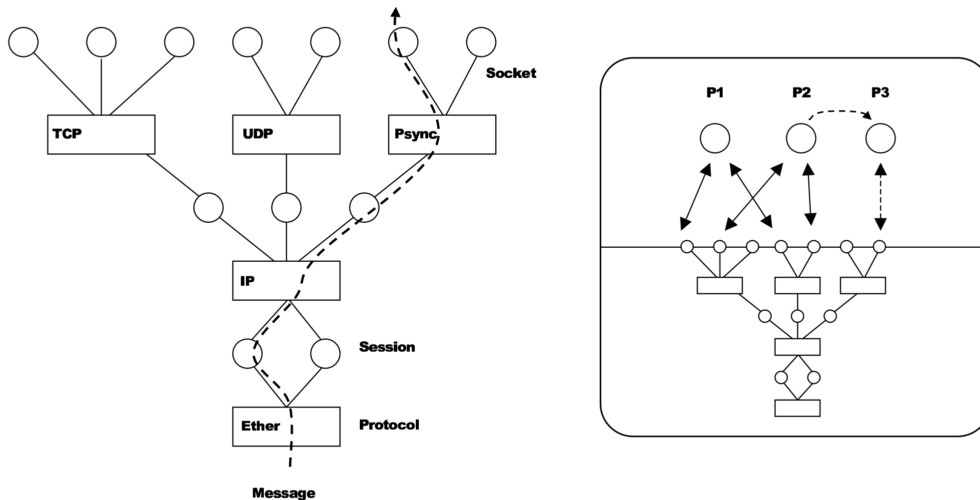


Figure 7.10: x-Kernel and its resource management

However, the whole picture looks different, as illustrated in Figure 7.10-right. Suppose that P1 has two sockets, one for a TCP connection, and the other for a UDP connection. Because each sockets are connected to different protocol processing modules, resource management of these connections is made independently, and there is no way to control the network I/O of P1. Similar problem happens in P2, which also has two connections. It is obvious that resource consumption of P1 and P2 is not partitioned. P3 is a child process of P2, which has a connection with Psync protocol. Now, the resource consumption of the connection is totally independent of P2, though we might want P3 to be bound by a resource granted to P2.

It is clear that there is no partitioning of resource consumption, nor resource management model. Indeed, this is another *horizontal* virtualization which just multiplexes the network interface into a flat structure of terminating sockets. This case suggests that the hierarchical packet scheduler and the hierarchical modeling of protocol processing have *completely different characteristics and implementation.* Indeed, they resemble each other just in their appearances.

VII.3.3 Hierarchical virtualization of network interface

The hierarchical virtualization model has been occasionally suggested to be an imitation of hierarchical packet schedulers, CBQ, and x-Kernel. However, the comparative studies

illustrated their distinctions. To understand the differences, it is beneficial to remember *sugar and salt*, which have similar appearance, but have completely different chemical composition and taste.

The conclusions drawn from the above studies are threefold. First, virtualization technologies can be classified by the classification scheme shown in Table 7.1. Second, *hierarchical virtualization* has unique property compared with other virtualization technologies in that it can affect virtualized entities both quantitatively and qualitatively. Third, the hierarchical virtualization of network interfaces greatly differs from other hierarchical virtualization schemes, although these hierarchical virtualization schemes have similar appearances.

Indeed, the hierarchical virtualization of network interfaces is a unique model for network I/O virtualization, in its hierarchical resource management capability. Such a virtualization would be applicable to other network systems. For example, proxy services and cache servers can be hierarchically virtualized for resource management.

VII.4 SUMMARY - "VIRTUALIZATION IS ADVANTAGEOUS"

This chapter attempted to break limitations of the computational model for network I/O programmability, the VIFlet model, presented in the last section.

First, the complete virtualization of network I/O, *via* extension of the VIFlet model, was attempted. A complete virtualization of network I/O is a virtualization that may replace all the network I/O processing. To this end, required capabilities were first investigated, and two technical complications were identified, namely, *write-access* and *packet creation capability.* The former may violate consistency of VIF routing and resource protection. The latter requires processing that must be done outside of VIFs. These complications are external to processing inside VIFs, and thus, named "externality in the VIFlet model ". Lastly, design choices are suggested, as an extension to the VIFlet model. One approach is to introduce an abstraction which handles system-wide issues. The other is to overload the VIFlet model to support such a service.

The second section investigated further improvement of network I/O performance. For

this purpose, each component technology of the network I/O system was reviewed, such as the Instruction Set Architecture, programming language, compiler, profiler, and optimizer. The review verified that it is reasonable to keep the programming model, VIFlet, and suggested the following approach. The program, written in the Java programming language, is compiled into a target code in an instruction set architecture more suitable for dynamic optimization, by a cross-compiler. Then, the kernel takes such a code as input, and dynamically translates it to a native binary, while applying simple load-time optimization. A userland profiler profiles the packet processing, performs code optimization during system idle time, and injects the optimized code back in to the kernel. The proposed scheme for virtualization of packet processing code would serve as a desirable framework for problems in modern operating systems and for further improvement of network I/O performance.

Lastly, comparative studies with other virtualization schemes were made. The review first illustrated the unique property of hierarchical virtualization approach among other virtualization schemes in that it affects target system both qualitatively and quantitatively. Then, the hierarchical virtualization of network interfaces model is compared against other known approaches of hierarchical virtualization, such as hierarchical CPU scheduling, Class Based Queuing, and x-Kernel. These systematic reviews illustrated the novelty and unprecedented properties of the virtualization model.

VIII CONCLUSIONS

This chapter concludes the dissertation on virtualization of network I/O in modern operating systems. First, conclusions are drawn from all the discussions made. Second, contributions of the thesis are briefly summarized. Third, possible future directions are suggested, followed by closing remarks.

VIII.1 CONCLUSION

In the research of network I/O, OS researchers have been mostly oriented toward Quality of Service issues as an application area of CPU scheduling technology. On the other hand, network researchers have been addressing flow of packets, without considering resource management inside end-hosts. Accordingly, the control model of network I/O on end-host operating systems has been in the gap between coverage of the research domains.

In this network age, network users and their applications are still unable to control their own traffic, even on their local host, not to mention in the network. Network processing code has been hardcoded inside operating system kernels, which formed boundaries between operating systems and inflated development and deployment cost of networking technology. As indicated, current network I/O systems are far from flawless, with respect to flexibility, resource management, access control, code portability, and performance. These issues are ascribed to the lack of appropriate control model in network I/O and hardcoding of packet processing code, which have severely retarded development of networking technology.

This dissertation addressed these problems, and proposed a *hierarchical virtualization of network interfaces*. This model accomplished flexible network control by users and appli-

cations, unlike existing systems designed solely for administrators, while providing resource protection. The prototype implementation demonstrated that users can efficiently and effectively control their network I/O, by dynamically injecting virtualized packet processing code.

The network I/O subsystem of modern operating systems has been incomplete. This research exemplified that virtualization technology can make up the incompleteness in network I/O of modern operating systems. In particular, our proposed hierarchical virtualization of network interfaces realized generality, flexibility, portability, and efficiency in network I/O control and implementation, suggesting that the hierarchical virtualization of network I/O can provide a general and decisive solution to problems in this domain, justifiable even as a design principle of modern operating systems.

VIII.2 CONTRIBUTIONS

VIII.2.1 Resource management model of network I/O

The proposed network I/O model, hierarchical virtualization of network interface, accomplishes a variety of properties desirable for virtualization of network I/O on end-host operating systems.

1. The hierarchical structuring coupled with the attachment mechanism realizes flexibility in control granularity. It supports control of network I/O at various granularity, such an independent flow and a set of flows, within a consistent framework.

2. It provides great freedom in the control of network I/O. Because it is a virtualization of a network interface, it can perform any control realized thus far on a network interface, such as bandwidth control, fair queuing, priority queuing, and security control.

3. The model provides a consistent mechanism for resource management and protection.

Because of this property, users and applications are allowed to perform any control within the limits of their allocated resources, unlike existing systems designed solely for administrators. Note that all the existing network control services are protected by privileged system calls to prohibit discretionary access from users.

It is amazing to realize that just a simple concept, hierarchical virtualization, can bring about reasonable solutions to difficult problems in the network I/O of end host operating systems.

VIII.2.2 VIFlet model for extensibility of network I/O code

Extensibility of packet processing code has been made through kernel APIs on operating systems. However, they provide direct access to kernel internals, without an abstraction layer, which requires substantial knowledge of kernel design and implementation. Accordingly, resulting programs become OS-dependent, and only skilled hackers of each OS can develop such software. The programs are also architecture-dependent, because they use native binaries. These factors severely compromise portability of network I/O software. Besides, although their native binaries look efficient, they cannot benefit from dynamic optimization technique, which is becoming increasingly popular.

The contributions of code virtualization are the opposite, as summarized below.

1. The VIF framework provides isolation of code execution and makes the in-kernel execution of user-supplied programs much safer. The code virtualization layer can provide further safety by performing runtime checking of security violation, in addition to the architecture independent executables. These properties realizes kernel extensibility for network I/O without administrator privilege.

2. The Java based VIFlet model provides *OS-independence*, and *flexibility in the degree of abstraction and information hiding*, through modification of the embedded class library.

3. The Java approach provides coding freedom to expert hackers, while offering kernel programmability even to entry-level programmers by hiding system details and providing code safety.

The code portability among operating systems and the architecture-neutral property would greatly reduce development and deployment cost of networking technology, which are limiting technical innovation in the field.

VIII.2.3 Prototype implementation and its profile

In addition to the contribution of the proposed model, the implementation efforts in Chapter IV and Chapter VI have provided valuable experiences, profile data, and implementations, which are also contribution of the research.

This research implemented the prototype implementation of the hierarchical virtualization model on FreeBSD 4 first, and ported the implementation onto several operating systems with completely different organizations, namely FreeBSD 5, OpenBSD, NetBSD, and Linux. This empirically proved that the virtualization model is implementable on ordinary modern operating system and warrants the portability listed above.

For these VIF kernels, many applications have been developed, such as `netnice`, `netniced` [102], extended `inetd`, extended `sh`, `login`, a network control module for Apache HTTP server, and an add-on to Firewall Builder [85]. There are also a variety of related tools: a testing framework of VIF functionality (`nndiag`), a traffic generator (`stream`), and firewall configurator (`vf`).

The VIFlet system might have an even greater contribution. This work produced an empirical proof that virtualized code does not jeopardize system performance. Indeed, because it is quite straightforward to run Java bytecode inside the kernel, the result suggests that further optimizations can make the virtualized code run even faster. The prototyped in-kernel lightweight Java Virtual Machine is highly extensible, and thus, can be reused for many other purposes. The Just-In-Time compiler would be far more usable than the interpreter, because of its performance advantage and compactness. As exemplified in Chapter VI, many tools

were also developed for the Java execution environment, such as `vifletttester`, `vifletd`, and the ROMizer, along with a variety of sample applications and benchmarks.

Please note that source code for all the programs is publicly open. Indeed, code contribution is enormous. Coupled with performance profiles and implementation lessons stated in this dissertation, it would be a basis for further research and production systems, in dynamic optimization of packet processing code.

VIII.2.4 Analysis of network I/O on modern operating systems

This dissertation analyzed and classified various models and implementations of network I/O on modern operating systems, which are also considered as contribution of the dissertation.

Chapter II presented a novel classification system of network I/O services based on their functionality (Figure 2.1), reusing *functional analysis* from Sociology. The classification contributed to draw requirements for the desirable network I/O service. That chapter also presented a classification system of packet processing programs, based on the degree of abstraction, and demonstrated the necessity of a model with flexibility in the degree of abstraction.

Likewise, Chapter VII presented another analysis of packet processing code, and illustrated properties of each process involved in packet processing, such as protocol processing and queuing control (Figure 7.1). That chapter also presented a classification system of virtualization technology (Table 7.1), which illustrates significance of hierarchical virtualization in general, among other virtualization technologies.

VIII.3 FUTURE WORK

Lastly, topics left as future work are listed below, including complete virtualization of network I/O, integration of the packet scheduler with the task scheduler, and end-to-end QoS.

VIII.3.1 Complete virtualization

First of all, it is an attractive option to extend the virtualization model to realize further flexibility in packet processing, allowing "complete virtualization" of network I/O code.

However, systematic investigation identified several complications in the complete virtualization of network I/O, namely, *write access* and *packet creation capability*. The former may violate the consistency of the VIF system. The latter necessitates processing external to the VIF system.

The last section suggested design choices for these problems. The write-access complication is addressed either by enforcing certain rules, or by ignoring changes made by the write operations. The externality is addressed either by aggressively overloading the VIFlet abstraction, or by devising another abstraction to handle all the external processing.

Although there are several design choices, it is still to be decided which one is most justified. Accordingly, it would be reasonable to first study several scenarios, such as in-kernel WWW servers and customized protocol stacks, to further investigate the properties of the model.

VIII.3.2 Integration with task scheduler

In the current implementation, scheduling of the VIF system and task scheduling of the underlying operating system are separated. Indeed, in the hierarchical virtual network interface framework, each VIF has its own packet scheduler, and it is independent of task scheduling of the operating system. This is because each invocation of VIF scheduling is made in event driven fashion, and guarantees its termination to the system. Accordingly, there has been little need for task scheduling in the VIF system.

However, once the VIFlet model is extended to support further flexibility of packet processing code, by the complete virtualization effort, there will be more need for task scheduling. Additionally, there is a need for QoS guaranteed service, as explored in [139, 59, 83, 126].

These QoS studies have been driven for quality assured execution of packet video applications, and enormous efforts have been made in this domain. On the other hand, task

scheduling of in-kernel virtual machines has rarely been addressed. Accordingly, there is a technical challenge worthy of further study.

VIII.3.3 End-to-end QoS

Because the network comprises intermediate-nodes and end-nodes, evolution of networking technology is only possible with coordinated advancement of both technologies. A clear illustration is IPv6, which required upgrading of router systems, and end-node systems. End-to-end QoS has exactly the same property in that it involves a variety of factors both in the network and on the end-nodes. They include control models, resource reservation schemes, signaling mechanisms, operating system support for guaranteed services, etc. It is clear that neither the network nor the end-nodes can appropriately address these problems alone. They need proper and close cooperation.

From a practical point of view, however, authority belongs to totally different parties in the global network: network service carriers and end-users. Accordingly, there will be no elegant solution in the end-to-end QoS problem. Even from a research point of view, it is obviously a laborious task, because such research involves too many factors. Because of these difficulties, although several efforts have been made in this domain, there has never been a decisive proposal. What happened was popularization of broadband access, together with extended backbone bandwidth, which just relieved the bandwidth problems we confronted. Nevertheless, that does not apply to wireless connections with limited bandwidth, which have become extremely popular. It is also obvious that broadband access does not cover the whole earth. Indeed, it will never happen that everybody has equal access, in reality. Resource constraint is a fundamental problem in this world, and thus, we need some solution for this problem anyway.

Virtualization unleashes services from underlying physical limits. The hierarchical virtualization is a generalized scheme for virtualization of end-host network I/O. Accordingly, this work would simplify the end-host factors in the problem domain. A counterpart of the end-node technology is a generalization of processing at intermediate notes, as pursued in the active network research [29].

Consequently, next reasonable step for this end-to-end problem is to combine both the approaches. In this regard, end-to-end QoS for medical networks would be good testbeds, as proposed in [100]. First, there is a solid need for end-to-end QoS; a transaction for critical cases such as Acute Myocardial Infarction must be prioritized over common cold cases. Second, it is well justified to make a medical network exclusive for certain purposes and this greatly simplifies the authority problem. For these advantages, a domain specific solution could be a breakthrough in the end-to-end QoS problem, which might contribute to the entire field of computer communication research. This is analogous to Artificial Intelligence field, where automatic diagnosis of infectious diseases provided such a breakthrough to the entire field [117].

VIII.4 CLOSING REMARKS

The dissertation is founded on an idealistic approach. It first defined an idealistic state of network I/O on the end-host operating system, and developed a virtualization model which meets the goal. Based on this model, a faithful prototype was implemented, and was ported onto several operating systems. On this foundation, many applications were developed, addressing actual problems in the real world. We may say that such research activity was based on *deductive logic*.

However, it is doubtful if such an idealistic approach could serve actual solutions in reality. In the field of computer systems, even the brightest idea can simply disappear from the market, if the marketing strategy is poor. In the PC processor market, CISC gained the hegemony. In the OS market, monolithic kernels are dominating. The market economy enforces a simple rule that a product which gains popularity is the winner, regardless of its orientation. Correctness of the approach is not endorsed a priori, but is later proved by the market share. Accordingly, the real world is governed by *inductive logic*.

Such market competition is considered to be cost effective, which optimizes resources on the earth. Nevertheless, it is also proved that the market is not always perfect. Indeed, it sometimes misbehave, as is observed in industrial pollution. The network I/O problem,

illustrated in the thesis, is such a "Market Failure" case. Although there have been a huge number of packet processing research projects in the field, almost everything is neglected, since they are not easily available in dominant operating systems. Meanwhile, because these experimental implementations do not have market share, the developers cannot afford to take all the trouble of porting the prototype onto production operating systems. As illustrated, there is a vicious circle, which causes stagnation in technical innovation.

This circle is reinforced by the nature of communication systems: communication involves independent systems governed by different parties with local interests, which inflates deployment cost. The difficulty is illustrated by the IPv6 case, which is still in the process of popularization, even with enormous support from governments of economic superpowers for a decade. Active Network research addressed problems at the network side, but has not yet produced any significant output with practical importance, because of the same reason. As a result, each implementation and research activity have ceased to address fundamental problems, targeting just niche markets formed around each operating system. If the market is incapable of solving such a vicious circle, it is probably the research community that must take the initiative. This thesis is one such attempt, and proved the significance of the virtualization approach in the problem domain. I envisage that this research will furnish a desirable foundation for further advancement of computer communication technology.

GLOSSARY

AOT : Ahead-of-Time compiler

An Ahead of time (AOT) compiler transforms a code into native machine code for future execution within a runtime environment.

Execution environment

An execution environment is a set of software that can execute programs written in certain instruction set. The term is analogous to Virtual Machine, but this term includes runtime library, although a VM typically does not. (see also: Virtual Machine).

Externality of VIFlet model

A property of VIFlets that causes problems which cannot be resolved within a VIFlet. For example, a VIFlet cannot perform protocol processing, because of the property. Other examples include routing of packets.

Instrumentation

The process of inserting generated statements into target code. The technology has been used mostly for measurements. In this dissertation, the term is used to insert generated packet filter code on the fly.

JIT : Just-In-Time compiler

A Just-In-Time compiler is a compiler that converts program code being run into native machine code on the fly.

JVM : Java Virtual Machine

A Virtual Machine that safely and compatibly executes the Java bytecode in Java class files. A Java Virtual Machine consists of a bytecode interpretation engine, a set of registers, a stack, a garbage-collected heap, and an area for storing methods. (see also: Virtual Machine).

leaf VIF

A leaf VIF is a VIF which is attached to terminating entities. (see also: VIF)

Network I/O processing

Network I/O processing is execution of certain class of software that handles data packets on a host machine.

Packet processing

Packet processing is execution of certain class of software that handles data packets on a host machine. This term is analogous to network I/O processing, but network I/O processing includes any accompanying processing needed for the handling, such as maintenance of a routing table.

Portability

Portability is a property of programs, either in binary form or in their source code, which assures execution of programs in different environments without modification. The dissertation illustrates that virtualization of network I/O realizes such portability in network I/O code, which comes in two forms, portability of OS code and application.

Protocol stack

A set of software components which is designed to appropriately deliver certain data from one place to another. This is a certain class of network I/O processing.

root VIF

A root VIF is the ancestor of all the VIFs in a VIF tree, connected to a network interface. (see also: VIF)

trunk VIF

A trunk VIF is a VIF which has a child VIF. (see also: VIF)

VIF : Virtual network interface

A VIF is an abstraction of a network interface. (see also: leaf VIF, root VIF, trunk VIF)

Virtualization

Virtualization is to modify the underlying details of a certain entity while keeping its predefined interfaces.

VM : Virtual Machine

Historically, the term has several meanings. Throughout this thesis, the term is used to denote a software component that can execute programs written in a certain instruction set.

Virtualization Ratio

Percentile of the virtualized portion in an entire packet processing system, including protocol processing and the user interface.

BIBLIOGRAPHY

[1] H. Abbasi, C. Poellabauer, G. Losik, K. Schwan, and R. West. A quality-of-service enhanced socket api in gnu/linux. In *the 4th Real-Time Linux Workshop*, December 2002.

[2] M. B. Abbott and L. L. Peterson. Increasing network throughput by integrating protocol layers. *Transactions on Networking*, 1(5):600–610, October 1993.

[3] A. Acharya and M. Raje. Mapbox: Using parameterized behavior classes to conne applications. In *the 9th USENIX Security Symposium*, 2000.

[4] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. Llva: A low-level virtual instruction set architecture. In *International Symposium on Microarchitecture*, pages 205 – 216. IEEE/ACM, December 2003.

[5] B. Ahlgren and P. Gunningberg. A minimal copy network interface architecture supporting ilp and alf. In *1st International Workshop on High Performance Protocol Architectures*, December 1994.

[6] U. Antsilevich, P-H Kamp, A. Nash, A. Cobbs, and L. Rizzo. IPFW. http://www.freebsd.org/.

[7] M. Bailey, B. Gopal, M. Pagels, L. Paterson, and P. Sarkar. Pathfinder: A pattern-based packet classifier. In *the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 115–123, November 1994.

[8] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM, June 2000.

[9] G. Banga, P. Druschel, and J. C. Mogul. Better operating system features for faster network servers. In *Workshop on Internet Server Performance*, June 1998.

[10] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans*, October 1999.

[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM 19th Annual Symposium on Operating System Principles (SOSP)*, October 2003.

[12] E. Bartel. New ttcp program. http://www.leo.org/˜elmar/nttcp/.

[13] A. Begel, S. McCanne, and S. Graham. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *SIGCOMM*, pages 123–134, 1999.

[14] S. Bellovin. Distributed firewalls. *;login:*, pages 39 – 47, November 1999.

[15] E. W. Biersack, E. Rfitsche, and T. Unterschiitz. Demultiplexing on the atm adapter: Experiments with internet protocols in user space. In *1st International Workshop on High Performance Protocol Architectures*, December 1994.

[16] N. L. Binkert, R. G. Dreslinski, E. G. Hallnor, L. R. Hsu, S. E. Raasch, A. L. Schultz, and S. K. Reinhardt. The performance potential of an integrated network interface. In *Advanced Networking and Communications Hardware Workshop (ANCHOR)*, June 2004.

[17] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-oriented full-system simulation using m5. In *the Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 2003.

[18] N. L. Binkert, L. R. Hsu, A. G. Saidi, R. G. Dreslinski, A. L. Schultz, and S. K. Reinhardt. Analyzing nic overheads in network-intensive workloads. Technical report, University of Michigan, December 2004.

[19] H. Bos and G. Portokalidis. Fairly fast packet filters. http://ffpf.sourceforge.net/.

[20] R. Braden, T. Faber, and M. Handley. From protocol stack to protocol heap - role-based architecture. In *First Workshop on Hot Topics in Networks (HotNets-I)*, October 2002.

[21] T. Braun. Limitations and implementation experiences of integrated layer processing. In *GI Jahrestagung*, pages 149–156, 1995.

[22] T. Braun, I. Chrisment, C. Diot, F. Gagnon, L. Gautier, and P. Hoschka. ALFred, an ALF/ILP protocol compiler for distributed application automated design. Technical report, INRIA Sophia Antipolis, January 1996.

[23] T. Braun and C. Diot. Protocol implementation using integrated layer processing. In *SIGCOMM*, pages 151–161, 1995.

[24] T. Braun and C. Diot. Automated code generation for integrated layer processing. In *Protocols for High-Speed Networks*, pages 182–197, 1996.

[25] T. Braun and C. Diot. Performance evaluation and cache analysis of an ilp protocol implementation. *IEEE/ACM Trans. Netw.*, 4(3):318–330, 1996.

[26] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *USENIX Annual Technical Conference*, June 1998.

[27] J. Brustoloni, E. Gabber, A. Silberschatz, and A. Singh. Signaled Receiver Processing. In *USENIX 2000 Annual Technical Conference*, pages 107 – 122. USENIX, June 2000.

[28] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating efficient protocol code from an abstract specification. *IEEE/ACM Trans. Netw.*, 5(4):514–524, 1997.

[29] T. M. Chen and A. W. Jackson (Editors). *Special Issue on Active and Programmable Networks, Network Magazine*. IEEE, July 1998.

[30] B. Cheswick and S. Bellovin. *Firewalls and Internet security: Repelling the Wily Hacker*. Addison-Wesley Longman Publishing, 1994.

[31] K. Cho. A framework for alternate queueing: Towards traffic management by PC-UNIX based routers. In *USENIX Annual Technical Conference*, pages 247 – 258. USENIX, June 1998.

[32] H. K. Jerry Chu. Zero-copy TCP in solaris. In *USENIX Annual Technical Conference*, pages 253–264, 1996.

[33] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *the ACM symposium on Communications architectures and protocols*, pages 200–208, September 1990.

[34] G. Combs. Ethereal. http://www.ethereal.com/.

[35] Symantec Corporation. Norton Personal Firewall. http://www.symantec.com/sabu/nis/npf/.

[36] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious server security. In *14th USENIX Systems Administration Conference (LISA 2000)*, 2000.

[37] A. Cox. The Linux traffic shaper. http://lwn.net/1998/1119/shaper.html.

[38] J. Crowcroft, I. Wakeman, and Z. Wang. Layering considered harmful. *IEEE Network Magazine*, 6(1):20–24, January 1992.

[39] W. Dabbous, S. O'Malley, and C. Castelluccia. Generating efficient protocol code from an abstract specification. In *SIGCOMM*, pages 60–72. ACM, August 1996.

[40] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, WA*, February 1996.

[41] P. Druschel, V. Pai, and W. Zwaenepoel. Extensible kernels are leading OS research astray. In *the Sixth Workshop on Hot Topics in Operating Systems (HotOS-VI)*, pages 38–42, May 1997.

[42] P. Druschel and L. L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. In *the 14th ACM Symposium on Operating Systems Principles*, pages 189–202, December 1993.

[43] K. Ebcioglu and E. Altman. Daisy: Dynamic compilation for 100 In *International Symposium on Computer Architecture (ISCA-97)*, pages 26–37. ACM, June 1997.

[44] A. Edwards and S. Muir. Experiences implementing a high performance tcp in user-space. In *the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 196–205, August 1995.

[45] T. Eicken and W. Vogels. Evolution of the virtual interface architecture. *Computer*, 31(11):61 – 68, November 1998.

[46] J. Elischer and A. Cobbs. The Netgraph networking system. http://www.elischer.org/netgraph/.

[47] D. Engler and M. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM*, pages 53–59, 1996.

[48] D. R. Engler. Vcode: a retargetable, extensible, very fast dynamic code generation system. In *SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI'96)*. ACM, 1996.

[49] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[50] M. A. Eriksen. Trickle: A userland bandwidth shaper for unix-like systems. In *USENIX 2005 Annual Technical Conference*, pages 61–70. USENIX, April 2005.

[51] R. Faulkner and R. Gomes. The process file system and process model in UNIX System V. In *the Winter 1991 USENIX Conference*, pages 243 – 252, 1991.

[52] D. C. Feldmeier and A. J. McAuley. Reducing protocol ordering constraints to improve performance. In *the IFIP WG6.1/WG6.4 Third International Workshop on Protocols for High-Speed Networks III*, pages 3–17, May 1992.

[53] M. E. Fiuczynski and B. N. Bershad. An Extensible Protocol Architecture for Application-Specific Networking. In *1996 Winter USENIX Conference, San Diego, CA*, pages 55 – 64, 1996.

[54] M. E. Fiuczynski, B. N. Bershad, R. P. Martin, and D. E. Culler. Spine: An Operating System for Intelligent Network Adapters. Technical report, University of Washington, 1998.

[55] M. E. Fiuczynski, R. P. Martin, T. Owa, and B. N. Bershad. Spine: a safe programmable and integrated network environment. In *ACM SIGOPS European Workshop*, pages 7–12, 1998.

[56] M. E. Fiuczynski, R. P. Martin, T. Owa, and B. N. Bershad. Spine: A Safe Programmable and Integrated Network Environment. In *8th ACM SIGOPS European Workshops, Portugal*, 1998.

[57] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Trans. on Networking*, 3(4):365 – 386, August 1995.

[58] The FreeBSD Project. http://www.freebsd.org/.

[59] S. Ghosh and R. Rajkumar. Network bandwidth reservation using the rate-monotonic model. In *International Conference on Software in Telecommunications and Computer Networks (SoftCOM 99)*, October 1999.

[60] S. Ghosh and R. Rajkumar. Resource management of the os network subsystem. In *5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 271–279, April 2002.

[61] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *the 6th Usenix Security Symposium*, 1996.

[62] P. Goyal, X. Guo, and H. M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *2nd Symposium on Operating System Design and Implementation (OSDI'96)*, pages 107 – 122. USENIX, October 1996.

[63] P. H. Gum. System/370 extended architecture: Facilities for virtual machines. *IBM J. Res. Develop*, 27(6):530 –544, 1983.

[64] D. Hartmeier. The OpenBSD Packet Filter. http://www.benzedrine.cx/pf.html.

[65] Bert Hubert. Linux Advanced Routing and Traffic Control. http://www.lartc.org/.

[66] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[67] S. Ioannidis, K. G. Anagnostakis, J. Ioannidis, and A. Keromytis. xpf: packet filtering for lowcost network monitoring. In *HPSR2002 (the IEEE Workshop on High-Performance Switching and Routing)*, pages 121–126, May 2002.

[68] S. Ioannidis, A. Keromytis, S. Bellovin, and J. Smith. Implementing a distributed firewall. In *ACM Conference on Computer and Communications Security*, pages 190–199, 2000.

[69] V. Jacobson, C. Leres, and S. McCanne. `tcpdump`. http://www.tcpdump.org/.

[70] P. G. Jain, N. C. Hutchinson, and S. T. Chanson. A framework for the non-monolithic implementation of protocols in the x-kernel. In *USENIX 1994 High-Speed Networking Symposium*, pages 13–30, August 1994.

[71] Java Embedded Microkernel (JEM) Project. http://dsg.port.ac.uk/projects/JEM/.

[72] M. B. Jones, D. Rosu, and M. C. Rosu. Cpu reservations and time constraints: Efficient, predictable scheduling of independent activities. In *the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 198–211, October 1997.

[73] M. Josefsson, J. Kadlecsik, H. Welte, J. Morris, M. Boucher, and R. Russell. Netfilter. http://www.netfilter.org/.

[74] M. Jung, E. Biersack, and A. Pilger. Implementing network protocols in java - a framework for rapid prototyping. In *International Conference on Enterprise Information Systems*, pages 649–656, 1999.

[75] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, and J. Jannotti. Application performance and flexibility on exokernel systems. In *ACM 16th Annual Symposium on Operating System Principles (SOSP)*, pages 52–65, October 1997.

[76] T. J. Killian. Processes as Files. In *Summer 1984 USENIX Confernece*, pages 203 – 207. USENIX, 1984.

[77] S.R. Kleiman. Vnodes: An architecture for multiplefile system types in sun unix. In *USENIX Summer Conference*. USENIX, 1986.

[78] A. Kleine. Tya. http://www.sax.de/˜adlibit/.

[79] B. Krupczak, M. H. Ammar, and K. L. Calvert. Implementing protocols in java: The price of portability. In *INFOCOM (2)*, pages 765–773, 1998.

[80] B. Krupczak, K. L. Calvert, and M. H. Ammar. Increasing the portability and reusability of protocol code. *IEEE/ACM Trans. Netw.*, 5(4):445–459, 1997.

[81] Oak Ridge National Laboratory. Parallel Virtual Machine. http://www.csm.ornl.gov/pvm/.

[82] Zone Labs. Zone Alarm. http://www.zonelabs.com/.

[83] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar. Predictable communication protocol processing in real-time mach. In *IEEE Real-time Technology and Applications Symposium*, pages 220–229, June 1996.

[84] Agnitum Limited. Outpost Personal Firewall Pro. http://www.agnitum.com/products/outpost/.

[85] NetCitadel LLC. Firewall Builder. http://www.fwbuilder.org/.

[86] K. Mackenzie, W. Shi, A. McDonald, and I. Ganev. An intel ixp1200-based network interface. In *Workshop on Novell Uses of System Area Networks at HPCA (SAN2-2003)*, February 2003.

[87] C. Maeda and B. N. Bershad. Protocol service decomposition for high-performance networking. In *ACM 14th Symposium on Operating Systems Principles (SOSP)*, pages 244–255, December 1993.

[88] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, pages 259–270, 1993.

[89] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications. In *the First IEEE International Conf. on Multimedia Computing and Systems, Boston, MA*, pages 90 – 99, May 1994.

[90] B. Metzler and I. Miloucheva. Design and implementation of a flexible user protocol interface. In *1st International Workshop on High Performance Protocol Architectures*, December 1994.

[91] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *11th ACM Symposium on Operating Systems Principles*, pages 39–51, November 1987.

[92] J. C. Mogul. Eliminating Receive Livelock in an Interrupt-driven Kernel. In *the USENIX 1996 Annual Tech. Conf., San Diego, CA*, October 1996.

[93] J. C. Mogul. Tcp offload is a dumb idea whose time has come. In *USENIX Workshop on Hot Topics on Operating Systems*. USENIX, May 2003.

[94] G. Muller, R. Marlet, C. Pu, and A. Goel. Fast, optimized sun rpc using automatic program specialization. In *The 18th International Conference on Distributed Computing Systems*, pages 240–249, May 1998.

[95] G. C. Necula and P. Lee. Safe kernel extenstions without run-time checking. In *the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI96)*, pages 229–243. USENIX, October 1996.

[96] Microsoft Developers Network. Windows sockets 2 documentation.

[97] P. Oechslin and S. Leue. Enhancing integrated layer processing using common case anticipation and data dependence analysis. In *1st International Workshop on High Performance Protocol Architectures*, December 1994.

[98] Okena. Stormwatch tm. http://www.okena.com/.

[99] T. Okumura, M. Moir, and D. Mossé. Netnice: nice is not only for cpus. In *The 9th International Conference on Computer Communication and Network (ICCCN2000)*. IEEE Communications Society, October 2000.

[100] T. Okumura and D. Mossé. A framework for semantics-based network management: Semantics aware internetworking with agent-oriented network programmability. In *ANTA2003 (The 2nd International Workshop on Active Network Technologies and Applications)*, May 2003.

[101] T. Okumura and D. Mossé. Netnice Packet Filter - Bridging the Structural Mismatches in End-host Network Control and Monitoring. In *INFOCOM2005*. IEEE, March 2005.

[102] T. Okumura, D. Mossé, M. Minami, and O. Nakamura. Quality of service manager for load-balancing clusters: An end-host retrofitting event-handler approach by netniced. In *The 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*. IEEE Communications Society, May 2003.

[103] S. O'Malley and L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[104] P. Patel, D. Wetherall, J. Lepreau, and A. Whitaker. Tcp meets mobile code. In *the Ninth Workshop on Hot Topics in Operating Systems (HotOS IX)*, May 2003.

[105] P. Patel, A. Whitaker, D. Wetherall, J. Lepreau, and T. Stack. Upgrading transport protocols using untrusted mobile code. In *ACM Symposium on Operating Systems Principles (SOSP 03)*, October 2003.

[106] I. Piumarta. The virtual processor: Fast, architecture-neutral dynamic code generation. In *Virtual Machine Research and Technology Symposium 2004*, pages 97–110, May 2004.

[107] N. Provos. Improving host security with system call policies. In *12th USENIX Security Symposium, Washington, DC*, August 2003.

[108] Ed. R. Braden. *RFC 2205: Resource ReSerVation Protocol (RSVP) – Version 1*. IETF, September 1997.

[109] H. S. Rahul, H. Balakrishnan, and S. Seshan. An End-System Architecture for Unified Congestion Management. In *the Seventh Workshop on Hot Topics in Operating Systems, Rio Rico, AZ*, 1999.

[110] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. B. Jones. Mach: a system software kernel. In *Proceedings of the 1989 IEEE International Conference (COMPCON)*, pages 176–178, San Francisco, CA, USA, 1989. IEEE Comput. Soc. Press.

[111] D. Reed. IP Filter. http://coombs.anu.edu.au/˜avalon/.

[112] D. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[113] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *Computer Communication Review*, 27(1):31 – 41, January 1997.

[114] R. Russell. Linux IP Firewalling Chains. http://www.netfilter.org/ipchains/.

[115] H. Schulzrinne and R. Lanphier. *RFC2326: Real Time Streaming Protocol (RTSP)*. The Internet Society, 1998.

[116] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *the 2nd Symposium on Operating Systems Design and Implementation*, pages 213–227, 1996.

[117] E. H. Shortliffe. *Computer Based Medical Consultations: MYCIN*. American Elsevier, 1976.

[118] C. Small and M. I. Seltzer. A comparison of OS extension technologies. In *USENIX Annual Technical Conference*, pages 41–54, 1996.

[119] GNU Software. gcj - The GNU compiler for the Java programming language, 2000. http://gcc.gnu.org/java.

[120] O. Spatscheck and L.L. Peterson. Defending Against Denial of Service Attacks in Scout. In *the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans*, October 1999.

[121] L. Stein. SBOX: Put CGI Scripts in a Box. In *USENIX Annual Technical Conference, General Track*, pages 145–155, 1999.

[122] J. Sugerman, G. Venkitachalam, and B.H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *the 2001 USENIX Annual Technical Conference*. USENIX, June 2001.

[123] Sun Microsystems, Inc. KVM white paper. May 2000. http://java.sun.com/products/cldc/wp/KVMwp.pdf.

[124] V. Sundaram, A. Chandra, P. Goyal, and P. Shenoy. Application performance in the qlinux multimedia operating system. In *the Eighth ACM Conference on Multimedia*, pages 127 – 136. USENIX, November 2000.

[125] D. Tennenhouse. Layered multiplexing considered harmful. In *IFIP WG 6.1/WG6.4 International Workshop on Protocols for High-Speed Networks*, pages 143–148, May 1989.

[126] Y. Tobe and H. Tokuda. Integrated qos management: Cooperation of processor capacity reserves and traffic management. *IEICE Trans. Comm.*, E81-B(11):1998–2006, 11 1998.

[127] W. Venema. TCP Wrapper: network monitoring, access control, and booby traps. In *UNIX Security Symposium III Proceedings*, pages 85 – 92. USENIX Assoc, September 1992.

[128] VMware. http://www.vmware.com/.

[129] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *ACM 15th Annual Symposium on Operating System Principles*, pages 40–53, 1995.

[130] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[131] D. A. Wallach, D. R. Engler, and M. F. Kaashoek. Ashs: Application-specific handlers for high-performance messaging. In *SIGCOMM*, pages 40–52. ACM, August 1996.

[132] M. Welsh. Safe and efficient hardware specialization of java applications. Technical report, University of California, Berkeley, May 2000.

[133] M. Welsh, A. Basu, and T. von Eicken. Incorporating memory management into user-level network interfaces. Technical Report TR97-1620, Cornell University, 1997.

[134] M. Welsh and D. Oppenheimer. User Customization of Virtual Network Interfaces with U-Net/SLE. Technical report, University of California, Berkeley, December 1997.

[135] M. Welsh, D. Oppenheimer, and D. Culler. U-Net/SLE: A Java-based User-Customizable Virtual Network Interface. In *Java for High-Performance Network Computing workshop, EuroPar '98,*, September 1998. SLE: Safe Language Extenstion.

[136] M. Welsh, D. Oppenheimer, and D. Culler. U-Net/SLE: A Java-based User-Customizable Virtual Network Interface. *Scientific Programming*, 7(2):147–156, 1999.

[137] R. Wild. Waba. http://www.wabasoft.com.

[138] P. Winterbottom and R. Pike. The design of the Inferno virtual machine. In *IEEE Compcon 97*, pages 241–244, February 1997.

[139] D. K. Y. Yau and S. S. Lam. Migrating sockets — end system support for networking with quality of service guarantees. *IEEE/ACM Transactions on Networking*, 6(6):700–716, 1998.

[140] M. Yuhara, B. Bershad, C. Maeda, and J. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *the USENIX Winter 1994 Technical Conference*, pages 153–165, January 1994.