2007

A New Packet Filter: Fast Packet Filtering by Exploiting CISC and SIMD ISA

Zhenyu Wu
College of William & Mary - Arts & Sciences

A NEW PACKET FILTER

Fast Packet Filtering by Exploiting CISC and SIMD ISA

Zhenyu Wu

Chengdu, Sichuan, China

Bachelor of Science, Denison Univerisity in Ohio, 2005

A Thesis presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Master of Science

Computer Science Department

The College of William and Mary
May, 2007

APPROVAL PAGE

This Thesis is submitted in partial fulfillment of
the requirements for the degree of

Master of Science

[illegible]

Zhenyu Wu

Approved by the Committee, April, 2007

[illegible]

Committee Chair
Assistant Professor Haining Wang, Computer Science

[illegible]

Chair and Associate Professor Phil Kearns, Computer Science

[illegible]

Assistant Professor Xipeng Shen, Computer Science

ABSTRACT PAGE

This thesis presents the New Packet Filter (NPF), a packet filter for high performance packet capture on commercial off-the-shelf hardware. The key features of NPF include (1) extremely low filter update latency with a strong security model for dynamic packet filtering; and (2) Gbps high speed packet processing. NPF achieves the former by employing a finite-state automata model as the pseudo-machine abstraction; and the latter by adopting CISC (Complex Instruction Set Computer) and SIMD (Single Instruction, Multiple Data) Instruction Set Architecture. The userspace library of NPF provides two sets of APIs (Application Programming Interfaces). One is to exploit the advantages of NPF in speed and security, while the other is for backward compatibility with existing BPF-based applications. We implement NPF in the latest 2.6 Linux kernel for both i386 and x86 64 architectures. We extensively evaluate its static and dynamic filtering performance on multiple machines with various hardware setups, and compare with BPF (the BSD packet filter), which is the de facto standard for packet filtering in modern operating systems, and optimized C filters that are used as the ceiling on performance. For static filtering tasks, NPF can be up to three times as fast as BPF; for dynamic filtering tasks, NPF can capture many more packets and data sessions than BPF, and is three orders of magnitude faster than BPF in terms of filter update latency.

TABLE OF CONTENTS

To my Mom and Dad, for their unending love and support.

ACKNOWLEDGEMENTS

I would like to thank Dr. Haining Wang, my research advisor, for his guidance to me in this thesis study. I would also like to thank Dr. Phil Kearns and Dr. Xipeng Shen for serving on my thesis committee, and thank the College of William and Mary IT department for supplying network traffic trace data.

This thesis could not be completed without help from Mengjun Xie, PhD candidate of Computer Science in College of William and Mary, and a good friend of mine. His collaboration on this research project is invaluable.

Finally, I would like to thank my parents, for their constant love and support, letting me to turn my interest in computers to a career in Computer Science.

LIST OF TABLES

LIST OF FIGURES

CHAPTER I.

INTRODUCTION

Packet filtering is a kernel facility for classifying network packets according to criteria specified by user applications, and conveying the captured packets from network interfaces directly to the designated userspace without traversing normal network stack. Since the birth of the seminal BSD Packet Filter (BPF) [13], packet filters have become critical infrastructure for network monitoring, engineering and security applications. In recent years, packet filters are facing intensified challenges posed by dramatically increasing network speed and escalating network application complexity. However, existing packet filter systems have not yet fully addressed these challenges in an efficient and secure manner.

Being the *de facto* packet filter on most of UNIX variants nowadays, BPF has shown insufficiency in handling both static and dynamic filtering tasks, especially the latter in high-speed networks [4], [6], [10]. A dynamic filtering task refers to the filtering process, in which some details used for describing the interested packets are not known *a priori* and can only be acquired during the monitoring process. As a result, the filtering criteria in dynamic filtering tasks have to be frequently updated throughout the process. A typical example of dynamic filtering tasks is to capture FTP passive mode data traffic[1]. A filter update in BPF

[1] The passive FTP file transfer mode is widely used today for its ability to work around the firewall and NAT on the client side.

must undergo a compilation and optimization process, in which the filtering criteria specified by human-oriented `pcap` filter language have to be translated into machine-oriented BPF instructions. This process could take milliseconds, even up to seconds, depending on the criteria complexity and system workload. For a high-speed network, hundreds or even thousands of packets may be missed by BPF during a filter update. The re-compilation process is extremely difficult, if not impossible, to avoid in that the design of the RISC (Reduced Instruction Set Computer) instructions and control flow optimization prevents applications from modifying BPF instructions directly. A workaround by collecting more packets than necessary, such as matching all TCP traffic in order to capture FTP passive data traffic, is very costly and not always practical for long-duration collection under heavy traffic load.

Inspired by the observation that the limited computing power of BPF[2] forces frequent context switches between kernel and userspace, recent packet filters such as xPF [10] and FFPF [4] move more packet processing capabilities from userspace into kernel to reduce context switches and improve overall performance. However, there is no good solution yet for reducing high filter update latency. Since xPF uses the BPF based filtering engine, it does not offer any improvement on filter update. FFPF attempts to solve this problem by using kernel space library functions, namely external functions, which on one hand avoid prolonged compilation, but on the other hand introduce security risks as well as programming complications. More specifically, external functions merge responsibilities of kernel filtering

[2] Such as not allowing loops within filter programs.

programs and userspace applications by loading pieces of pre-compiled binaries from userspace and executing them inside kernel. Such external functions are exposed to potential security risks because they are typically written in “unsafe” languages (such as C and assembly). In addition, coding an external function is essentially kernel programming. Comparing to userspace programming, kernel programming has higher restrictions on resource management, lower access protection, fewer library support, and no powerful debugging tools, making it more difficult to complete and prone to error.

In this thesis, we propose the New Packet Filter (NPF) to achieve superior performance on both static and dynamic packet filtering, while retaining the simplicity and elegance in BPF. Based on the pseudo-machine abstraction and filtering language primitives of BPF, NPF renovates the filtering engine design by employing CISC (Complex Instruction Set Computer) and SIMD (Single Instruction, Multiple Data) instruction set architecture. With carefully re-designed Instruction Set Architecture (ISA) and employment of Deterministic Finite Automata (DFA) computation model, NPF eliminates the necessity of compilation and security checking during filter update, resulting in significantly reduced filter update latency and increased packet filter execution efficiency.

We implement NPF in the latest Linux 2.6 kernel on both i386 and x86 64 platforms. The kernel implementation of NPF is fully compatible and can coexist with BPF. The corresponding NPF user-level library not only provides a BPF-primitive-based filter programming interface, but also includes a BPF compatible compiler, making transition from

BPF fast and easy. To validate the efficacy of NPF, we conduct extensive experiments on four machines with different hardware setups and computing power. For static filtering tasks, NPF runs as fast as BPF on simple filtering criteria, but is up to three times as fast as BPF on complex filtering criteria. In comparison to optimized C filters, which are used as the performance ceiling, NPF performs much closer to the optimized C filters than BPF. For dynamic filtering tasks, NPF can lower the filter update latency by three orders of magnitude and capture up to 288% more packets than BPF under high traffic rates.

The remainder of this thesis is structured as follows. Chapter II surveys related work on packet filtering. Chapter III details the design of NPF. Chapter IV describes the implementation of NPF. Chapter V evaluates the performance of NPF. Chapter VI concludes this thesis.

CHAPTER II.

RELATED WORK

The novel concept of *Packet Filter* is first introduced in [15], where the CMU/Stanford Packet Filter (CSPF) is proposed as a kernel-resident network packet demultiplexer. In essence, CSPF provides network packets a fast path for reaching their destined userspace applications, in comparison to the normal layered/stacked path in the kernel. Thus, the literal meaning of filtering in CSPF is *de-layered demultiplexing* [22]. The original motivation behind CSPF is the necessity of providing a mechanism to facilitate the implementation of network protocols such as TCP/IP at userspace. Although the purposes and techniques vary in subsequent packet filters, the CSPF model of residing in the kernel and being protocol-independent is inherited by all its descendants.

Different from CSPF, the BSD Packet Filter (BPF) [13] aims to support high-speed network monitoring applications such as tcpdump [11]. Under the architecture of BPF, userspace applications inform the in-kernel filtering machine their interested packets by means of predicate based filtering language [14] and then receive from BPF the packets that conform to the filtering criteria. To achieve high-performance packet filtering, BPF employs the following novel techniques: (1) in-place packet filtering to reduce unnecessary cross-domain copies, (2) the register-based filter machine to fix the mismatch between the filtering program and its underlying architecture, and (3) the Control Flow Graph (CFG) model to avoid unnecessary computations in the Expression Tree model. BPF+ [3] further enhances the performance of BPF by exploiting global data-flow optimization to eliminate redundant predicates across filter criteria and employing just-in-time compilation to convert a filtering criterion to native machine code. xPF [10] increases the computation power of BPF by introducing persistent memory and allowing backward jumps.

Following the avenue of packet demultiplexing for assisting user-level network protocol coding, MPF (Mach Packet Filter) [23], PathFinder [2], and DPF (Dynamic Packet Filter) [7] have been proposed successively. To efficiently demultiplex packets for multiple user-level applications and to dispatch fragmented packets, Yuhara, *et al*. developed MPF [23] by extending the instruction set of BPF with an associative match function. With the same goal of achieving high filter scalability as MPF, PathFinder [2], however, abstracts the filtering process as a pattern matching process and adopts a special data structure for the abstraction. The abstraction makes PathFinder amenable to implementation in both software and hardware and capable of handling Gbps network traffic. Sharing the same abstraction with PathFinder but focusing particularly on high performance, DPF [7] utilizes the dynamic code generation technology, instead of a traditional interpreter-based filter engine, to compile packet filtering criteria into native machine code.

Due to the inefficiency of handling dynamic ports by tcpdump, a special monitoring tool mmdump [21] has been developed to capture Internet multimedia traffic, in which dynamic ports are widely used. Being an extension of tcpdump, mmdump does not provide a generic and efficient solution to dynamic filtering tasks. Complementary to software-based packet capture solutions, multiple hardware-based solutions such as [5], [17] have been proposed to meet the challenge posed by extremely high speed networks. Moreover, other than the packet filter based network monitoring architecture, there exist many specialized-architecture monitoring systems such as OC3MAN [1], Windmill [12], NProbe [16], and SCAMPI [19]. Even with these hardware or specialized system solutions, packet filters will still play a major role in network monitoring and measurement, due to its simplicity, universal installation, high cost-effectiveness, and rich applications.

The Fairly Fast Packet Filter (FFPF) [4] is the most recent research on packet filtering. FFPF greatly expands the filter capacity by using external functions which essentially are native code running in kernel space. It invents the concept of flow group, in which several monitoring programs share the same packet buffer, thus eliminating redundant packet copying. In addition, it features language neutral design, provides compatibility to BPF, and offloads filter processing to network interface cards with a Network Processing Unit. As the most visible packet filter research since BPF, FFPF achieves significant performance improvement. However, FFPF and NPF resolve quite different problems. FFPF focuses on the packet filtering framework and its main contribution lies in the improvement of scalability for multiple simultaneous monitoring programs; while NPF aims at the packet filtering engine and provides a fast, flexible, and safe filtering mechanism for the applications. Since FFPF's framework is language neutral, NPF and FFPF are complementary to each other.

CHAPTER III.

NPF DESIGN

In this chapter we first state the motivation of NPF, describe our design approach and tradeoffs, and then present the detailed ISA. Finally, we analyze the characteristics of NPF in terms of performance, security, and computational power.

i. Motivation

The inefficiency of BPF observed in our past experiences directly motivates the design of NPF. The most observable performance degradation of BPF, as mentioned before, emerges in dynamic filtering tasks. This degradation is attributed to the compilation process in filter update, which has been pointed out in previous research such as [6] and [21].

The filter re-compilation process could be unduly long when the BPF filter is capturing traffic on high-speed networks. Frequent filter updates, often required by the filtering task, exacerbates the performance degradation. The duration of filter update in BPF would be significantly reduced if the re-compilation phase were skipped. This trick is possible for applications with very limited purpose such as `mmdump` [21]. However, for general purpose network monitoring applications, such as NIDS, filter re-compilation is inevitable, due mainly to the following two reasons. First, with a RISC-like pseudo-machine filtering engine, each `pcap` language primitive is implemented as a variable number of simple BPF RISC instructions. Changing one of the primitives without recompiling often mandates updating instructions, such as branch target address, throughout the entire compiled filter, which could be more costly than re-compiling. Second, BPF employs control flow optimization in order to speed up filter execution; however, this technique also renders direct instruction-level filter update impossible. Control flow optimization merges several identical groups of instructions into one, reduces both filter program size and execution time. However, if subsequent changes in one control flow require updating a group of instructions shared by other control flows, direct instruction modification will incur errors in the filter. The correct updating, which requires control flow analysis, instruction generation and patching, is almost certainly much more costly than recompiling.

Upon further inspection of BPF, we discover that on one hand the RISC ISA gives BPF flexible programming and easy analysis; on the other hand, it makes the filter execution very inefficient. Each BPF instruction accomplishes a very simple task, such as loading, arithmetics, or conditional branching. However, in order to execute an instruction, the pseudo-machine engine has to complete an execution cycle, which includes at least one load, one or more arithmetic operations, and one or more conditional branches. Therefore, the useful processor cycles spent in evaluating the actual packet filtering criteria only make up a small fraction of the total; most of the time the processor spins "uselessly" inside the BPF filtering engine.

ii. Design Approach

Despite the observed performance inefficiency, BPF is still the most widely used packet filter. Its great success is mainly attributed to (1) the generic pseudo-machine abstraction, which guarantees cross-platform compatibility; and (2) its natural language like primitive based filter language, which ensures the ease of use to network administrators. Therefore, we decide to inherit the pseudo-machine abstraction and the filter language primitives from BPF, while developing our own filtering model that resolves the problems mentioned above.

The primary objective of NPF is to achieve low filter update latency. Our approach to reaching this goal is by eliminating filter re-compilation and security checking – the two most time-consuming stages in filter update.

The process of compilation is essentially a translation from a high level program description to a low level instruction collection, which the underlying hardware can understand and execute. We avoid compilation in NPF by abstracting and specializing our pseudo-machine instructions, establishing a CISC-like architecture that maps the high level descriptive filter language elements (primitives) directly to low level machine operations. In other words, each filter primitive has a corresponding pseudo-machine instruction; if the description of filtering criteria is given in compliance of the NPF ISA, the description itself is directly executable by the filtering engine without any translation (compilation). The design rationale behind creating a CISC-like ISA lies in the observation that packet filtering is a very specific system tool with limited operations. The `PathFinder [2]` research shows that all operations in packet filtering can be generalized as pattern matching, and our study further reveals that a limited set of pattern matching operations can cover all common packet filter operations. As a result, although NPF has a very specialized instruction set, which includes only 22 different type of instructions, it is able to perform the equivalent operations to any `pcap` language primitives (except arbitrary arithmetic) used in BPF.

The simplicity-driven principle also leads to the simplified computational model, Deterministic Finite Automata, which enables NPF to maintain security without filter program security checking. Because of the highly specialized ISA, each NPF instruction is able to perform data loads, (predefined) arithmetic calculations, and comparisons. The execution path of the filter program is determined by the Boolean evaluation result of each instruction: either continue ("`true`") or abort ("`false`"). Thus, we simply do not provide the filter program with any storage, nor do we allow the program to control its execution path with branch instructions. With the fixed set of instructions, acyclic execution path and zero data storage, any NPF filter program is safe to run in kernel. To further optimize filter update, we define all NPF instructions with fixed length, so that any instruction can be quickly located and modified.

The secondary objective of NPF is to increase filter execution efficiency. The use of CISC-like ISA already opens a door for NPF performance optimization. To further boost the speedup, we introduce SIMD[3] processing for nearly all NPF instructions. SIMD allows the processor to perform a single instruction interpretation and apply the same operation on many sets of data, thereby significantly reducing the instruction interpretation cost.

[3] SIMD has been widely used in contemporary high performance processors, such as Intel Pentium series and IBM Power series processors.

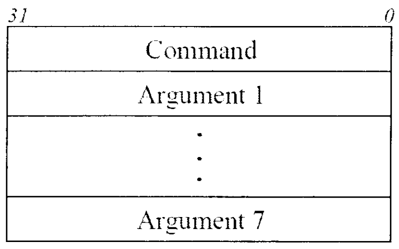


Figure 1: The NPF Instruction Format

iii. Detailed ISA

Like BPF, the basic building block of NPF is instruction. Figure 1 illustrates the format of an NPF instruction. As mentioned above, for the purpose of fast instruction address during filter update, all NPF instructions have the same size: one 32-bit command field and seven 32-bit parameter fields.

We formulate our instruction set based on BPF primitives. We first classify BPF primitives into two categories according to their address modes. "*Direct addressing*" primitives, such as "`ether proto`" and "`ip src host`", fetch data from an absolute offset in a packet. "*Layer 1 indirect addressing*" primitives, such as "`tcp dst port`", address data by calculating variable header length of a protocol layer and adding a relative offset to it. We then generalize the manipulation and comparison operations used in each layer. There are three basic operations: (1) *test if equal*, (2) *mask and test if equal*, and (3) *test if in range*. There also exist some variations in each layer, such as the operand width. Finally, we design the complex instructions to accomplish the corresponding operations. We come up with 14 different operations that, alone or by combination, are able to perform equivalent operations of any BPF primitives (except "`expr`" which involves arbitrary arithmetic).

To further exploit CISC architecture and enhance performance, we introduce a new addressing mode, "*Layer 2 indirect addressing*", with 4 additional instructions. In the new addressing mode, filtering operations address data by first performing "*Layer 1 indirect addressing*" to retrieve the related information, which is used to calculate variable header length of the higher layer, and then adding a relative offset. Note that BPF does not provide such primitives, but there are real demands, such as filtering based on TCP payload. Moreover, we add 4 more "power instructions" that performs equivalent operations of several frequently used BPF primitive combinations, such as "`ip src and dst host`", "`tcp src or dst port`". Therefore, in total the NPF ISA consists of 22 different types of instructions.

Table 1 is a sample of the NPF instruction set, which captures the characteristics of NPF CISC-like ISA. The first (the leftmost) column shows three addressing modes; the second column shows different types of instructions; the third column explains their functionality, and the fourth column shows the equivalent operations in BPF. Although the NPF instructions are considered to be "specialized", they are still fairly generic in that given different parameters; one NPF instruction can function as several different `pcap` language primitives, as listed in some cells of the fourth column. Based on the NPF instruction set, we can derive alternative faster implementations for some BPF primitives. For example, the "`ip tcp port`" primitive in BPF requires examining whether a packet is IP, non-fragment, and TCP in three separate steps with six instructions. In NPF, we can take advantage of the "continuous masked comparison" instruction (`D_LEQ_M`), performing the same examinations in a single instruction.

Addressing mode	Opcode	Semantics	Corresponding BPF operation
Direct Addressing (Offset specified directly)	D_EQ	Test if a 32bit integer at given offset equal to an integer in parameter list	Compare host IP address
	D_MEQ	Similar to the above, but a bitmask is applied to both comperands before comparison	Compare packet protocol; Compare host IP netmask
	D_DMEQ	Perform the same operation above on two adjacent 32bit integers, reutrn true if any one matches	Compare source or destination IP netmask
	D_LEQ_M	Test whether up to three continuous 32bit integers applied with separate bitmasks equal to the supplied operands	Compare source and destination IP netmask; Tell if a packet is IP, non-fragment, and TCP/UDP
Layer 1 Indirect Addressing (Offset calculated by resolving a variable layer header length)	L1_SEQ	Test if a 16bit shortint at given offset equal to a shortint in parameter list	Compare IP protocol; Compare TCP / UDP port
	L1_SSEQ	Perform the same operation above on two adjacent 16bit shortint, reutrn true if both matches	Compare source and destination TCP/UDP port
	L1_SRNG	Test if a 16bit shortint at given offset within a numerical range specified by the parameter list	Capture TCP / UDP packets on a range of ports
Layer 2 Indirect Addressing (Offset calculated by resolving two variable layer header lengths)	L2_LEQ	Compare up to five 32bit integer from given offset with parameter list	TCP payload content matching
	L2_LEQ_M	Compare up to two 32bit integer from given offset with parameter list, each one has separate bitmask applied before comparison	TCP payload content matching (bit precision)

Table 1: Sample of NPF Instruction Set

We also add the SIMD feature into our instructions set by packing additional operands into unused parameter list. For example, the "`Direct addressing load, test if equal`" instruction (`D_EQ`) uses only one 32-bit operand; while the SIMD version of this instruction can carry up to 6 additional operands, and the corresponding operation becomes "`Direct addressing load, and test if equal on op[0], or op[1], or op[2], ... op[6]`".

A series of instructions connected by logical "AND" form a *Pass*. In essence, a *Pass* is a single complete control flow. As the name "*Pass*" denotes, one can think of it as a passage from the network into the application. When a packet arrives at a *Pass*, instructions are evaluated one-by-one. If all evaluation results are "`true`", the packet is accepted and copied to the userspace; otherwise, if any evaluation result is "`false`", the packet fails the current *Pass*, and will be tested by remaining *Passes* or simply dropped. For fast addressing, each *Pass* includes 30 instructions. With the support of complex SIMD instructions, the 30 instructions can express a very large and complex control flow. For example, in Chapter IV the static filtering criteria No. 6 can be expressed using only 10 distinct NPF instructions[4].

Multiple control flows are supported in NPF by a collection of *Passes* called a *PassSet*. At maximum, a *PassSet* can include 127 *Passes*. The relationship between *Passes* is exemplified in Figure 2. A new *Pass* can be created from scratch or incrementally by the

[4] Two *Passes* are used to express the filtering criteria completely, each containing 8 instructions, but sharing 6 common instructions.

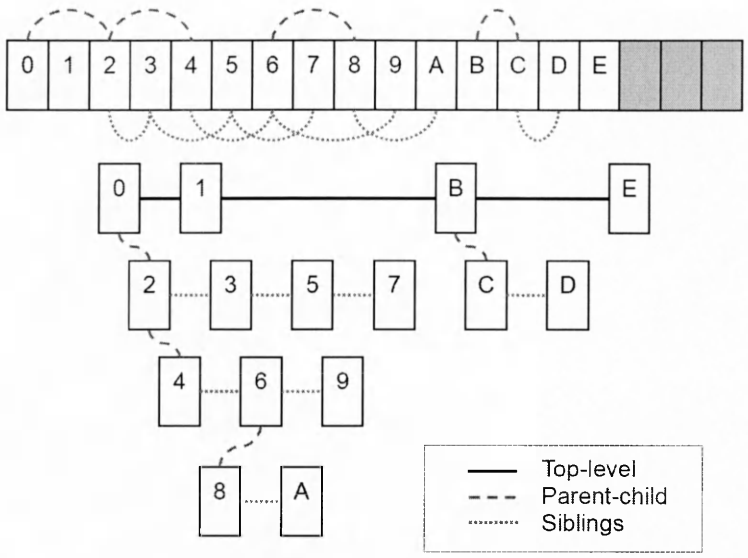


Figure 2: NPF Passes Relation Diagram

"duplication" procedure. Normally, the top-level *Passes* are coined afresh while all lower level *Passes* are formed by duplication. The *Pass* being duplicated is the "*parent*" and the duplication product is the "*child*"; those *Passes* that share the same *parent* pass are "*siblings*". The *Passes* are evaluated in a depth first manner, and a packet either succeeds in one `Pass` or fails all `Passes`.

iv. Design Tradeoffs

There are two design tradeoffs in NPF. First, by eliminating the compilation stage, NPF also forfeits possible instruction-level optimization. The compilation and optimization in BPF build an acyclic CFG (Control Flow Graph) of all given filtering criteria, and merge the common criteria across control flows to achieve optimal performance. Without such a stage, the NPF filter program instead is simply a "Control Flow Collection", in which the Boolean expressions of filter criteria are in disjunctive normal form. To avoid the potential performance pitfall, NPF employs a different optimization method called “passive incremental optimization”. It originates from our observation that throughout a dynamic filtering task, the set of criteria newly-added to the filter program are very likely related to existing criteria. For example, the newly added set of criteria often monitors the same host but on different ports or the same protocol but on different hosts. NPF utilizes this “incremental” relationship between criteria to remove redundant instruction evaluations. However, due to latency constraints, this optimization cannot be done automatically on every filter update. Instead, it relies on the application to call a special “duplication” function to signify the incremental relationship. When duplicating a set of criteria (the *parent*), NPF creates a copy of the criteria (the *child*) but leaves a special flag on each individual criterion indicating whether it has been modified. If any subsequent changes to the *child* occur, “unchanged” flags are cleared accordingly. NPF always evaluates the *children* after the evaluation of their *parent*. When evaluating a *child*, the filtering engine only evaluates the criteria that are either modified, or have not been evaluated in the *parent*.

The second design tradeoff is redundant computation in the sub-instruction level. Because NPF does not provide any storage to the filter program, no information can be explicitly exchanged even between instructions. As a result, instructions must maintain a certain computation redundancy. For example, NPF filter programs cannot compute and store the protocol header length, such as IP header length. Thus, each instruction that handles packet payload must perform a header length calculation. On one hand, this feature is desired since NPF instructions are designed to be independent of each other and thus can be modified in arbitrary fashion without affecting the operations of instructions executed afterwards. On the other hand, when handling a large number of instructions, the redundant computation is a waste of CPU cycles. We resolve this problem by introducing a computation cache, an implicit mechanism to share information across instructions. It functions as a hint to the pseudo-machine. Upon the arrival of a new packet, the computation cache is reset; and before calculating the protocol header, the filtering engine first checks whether the corresponding computation cache holds a valid value. If so, that value is returned directly from the cache without performing the actual calculation. Otherwise, calculation is performed and the result is recorded in the cache. Thanks to the computation cache, the protocol header length calculation is performed at most once per packet.

v. Analysis

Before giving detailed analysis of NPF in terms of performance, security, and computational power, we first summarize the shared design principles of NPF with other packet filters, especially BPF, as well as its unique design features that distinguish NPF from other packet filters. The shared and unique design features are listed as follows, in which the shared are marked with "+" and the unique are marked with "*".

- Runs as a kernel module, filtering packets "in place". (+)
- Follows the "Acyclic Control Flow" model, avoiding unnecessary computation. (+)
- Uses architecture-independent pseudo-machine instructions, making filter codes

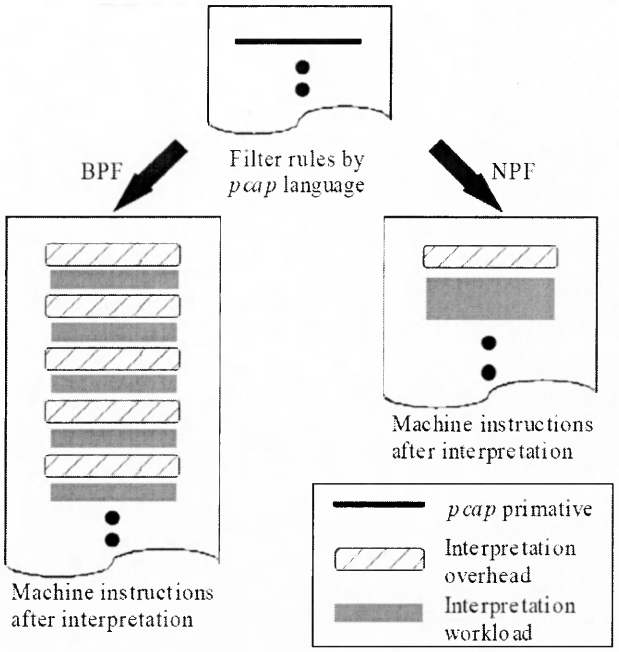

BPF
Filter rules by *pcap* language
NPF
Machine instructions after interpretation
Machine instructions after interpretation
pcap primative
Interpretation overhead
Interpretation workload

Figure 3: Overhead Comparison between BPF and NPF

cross-platform. (+)

- Utilizes CISC and SIMD instruction set architecture, eliminating the compilation stage. (*)
- Supports in-place and incremental filter modification. (*)
- Simplifies the computational model, ensuring security without security checking. (*)

a. Analysis of Performance

The performance superiority of NPF mainly comes from three aspects: its ISA, SIMD instructions, and low filter update latency.

The CISC-like ISA dramatically speeds up filter execution time by reducing the high cost of pseudo-machine instruction interpretation. As shown in Figure 3, the execution of a `pcap` language primitive requires several (say *N*), RISC-like instructions in BPF, while it only takes one CISC-like instruction in NPF. Thus, the ratio between the interpretation overhead of BPF and NPF is *N* to 1. As described before, the number *N* is normally larger than 3, and sometimes can be as large as 7 to 10.

Because of the pseudo-machine software execution overlay, the CISC-like architecture not only no longer suffers from hardware obstacles[5], but also introduces more opportunity for implementation and run-time optimizations, which are not possible for RISC-like pseudo-machines. Reduced instructions only perform minimum operations. Their implementations are essentially discrete instructions scattered across the pseudo-machine code body, so it is hard for human or compilers to optimize. While the underlying hardware is often optimized to execute a "continuous logic flow" by utilizing the spatial and temporal program localities, the execution of RISC pseudo-machine is two logic flows scrambled together. The logic continuance of the filter program operations is brutally interrupted by the pseudo-machine state maintenance code throughout the execution. Therefore, the run-time optimization provided by the hardware cannot be well utilized by RISC architecture.

The NPF complex instructions are implemented as a set of closely-related operations integrated together: load data from memory, manipulate, compare, and branch. The integrated operations can be easily optimized by hand or compilers. For example, in NPF the x86

[5] The key advantages of RISC architecture, such as simpler hardware implementation, more registers, and better pipeline performance, does not apply here any more.

processor's general registers can be utilized to speed up the filter program execution, which is not possible in BPF. The execution of CISC-like pseudo-machine is friendly towards the underlying hardware. Although there are still two logic flows, they interleave "naturally", in other words, the filter program executes continuously until reaching a branch point, then the pseudo-machine state maintenance code kicks in, starting the execution of the next block of filter program, and so on.

SIMD instructions further boost NPF's performance by amortizing the interpretation cost of an instruction over multiple operands. As an example, when comparing the source port of a TCP packet against multiple values, BPF requires one instruction per port value; however, NPF can pack as many as 12 port values into the SIMD version of the `L1_SEQ` instruction.

For a dynamic filtering task, NPF provides a set of system calls for adding, removing or modifying criteria. Instead of updating filter descriptions in userspace, recompiling, and performing kernel security checks, NPF simply pin-points the *Pass* and *Instructions* to be updated. Even during the process of modification, only the target *Pass* is affected while all other *Passes* continue to function as usual, allowing the matched packets to be accepted.

b. Analysis of Security

Security has always been a concern in the design of a packet filter. More specifically, the security here refers to the correctness of filtering codes, since modern packet filters are executed in kernel address space. Without proper security checks, a faulty filter program containing dead loop, wild jump, array index out of bound, etc. could lead to unpredictable results. A maliciously crafted filter program can bypass any user level access protection and become a serious security hole.

Depending on the design model, different packet filters have different mechanisms to enforce the security of the filtering programs. For example, thanks to its pseudo-machine abstraction, BPF performs a security check in the kernel just before the filter program is attached, and rejects any program having backward jump instructions. The FFPF filter languages allow memory allocation; therefore FFPF has compile-time checks to control resource consumption and run-time checks to detect array bound violations.

General security principles suggest that any program should not have more privileges than its functional needs. Essentially, the security checks in BPF and FFPF are the efforts to limit certain privileges (computational powers) that the filter program does not need. Therefore, we attempt to completely remove the extra computational powers from the basic model and design in NPF.

As a result, NPF uses the Deterministic Finite Automata (DFA) computational model as its pseudo-machine abstraction, instead of Random Access Machine model in BPF and FFPF. The benefit of using DFA computational model is that during the execution of an arbitrary program, the state of the machine will always be within a predetermined set. In other words, no program will be able to perform any unspecified operation. With this computational model, an NPF program is “well-behaved” and security checks are deemed unnecessary.

c. Analysis of Computational Power

Compared to RAM, DFA has reduced computational power as a tradeoff for stronger security. However, we would like to argue that this reduction is harmless in the context of packet filtering. The complete implementation of all (except arbitrary arithmetic) `pcap` language primitives in NPF, successfully proves that DFA has sufficient computational power for a network packet filter to perform well defined tasks. In addition, the loss of ability to perform arbitrary arithmetic has negligible impact on NPF's usage or performance for the following two reasons.

Firstly, existing NPF instructions can cover all aspects of general purpose packet filtering. For instance, the masked integer comparison instructions are applicable to a large range of operations, from protocol identification to packet content matching. Although currently only used for a limited number of protocols (IP and TCP), the layered packet header resolution instructions can easily be generalized to apply to other protocols.

Secondly, NPF's complex instructions are, in fact, very good at handling arithmetics, as long as the operations are well defined. If needed, an NPF instruction can perform more complex arithmetics than any `pcap` language primitives, with orders of magnitude faster speed. Since most of commonly used packet filtering criteria are very well defined, existing NPF instructions are fully capable of handling the corresponding sequences of arithmetics. For emerging filtering criteria that require new arithmetic operations, we could easily add new instructions and implementations if necessary.

CHAPTER IV.

IMPLEMENTATION

We have implemented the NPF kernel filtering module for the latest Linux 2.6 kernel as well as the supporting userspace libraries. Currently we provide implementations for both i386 and x86_64 architectures, and we plan to port NPF to other open-source UNIX variants such as FreeBSD in the future.

i. Kernel Implementation

The Linux Socket Filter (LSF) in Linux kernel is the module equivalent to BPF for BSD UNIX systems. The core filtering engine is implemented in file *filter.c* in the "*net/core/*" directory. It provides a function call interface `sk_run_filter()`, through which the filter program is executed and a decision whether or not a packet is accepted is made. A special Linux socket called `AF_PACKET` is used for communication between userland applications and LSF module. Userland applications make use of LSF by first creating an `AF_PACKET` socket, and then attaching LSF filter program to the socket through system call `setsockopt()`. Normally, these basic application-kernel interactions are handled by the userland supporting library `libpcap`. Refer [8], [9] for the details of LSF internals.

To maximize the compatibility with the existing framework, we also implement the core filtering engine of NPF in a single file *npf.c* in the "*net/core/*" directory, providing the identical function call interface `sk_run_npf()` in parallel with LSF. We also add the NPF user-kernel

Routine	Functionality
NPF_Open	Create and attach an NPF *PassSet* to a socket
NPF_NewPass	Get a new *Pass* from the *PassSet*
NPF_DelPass	Remove a *Pass* from the *PassSet*
NPF_DupPass	Perform a duplication of a given *Pass*
NPF_SelectOp	Select a pre-defined operation (equivalent to a *pcap* language primitive) into an *instruction* of a given *Pass*
NPF_AddParam	Add an additional (SIMD) parameter to an *instruction* in the given *Pass*
NPF_DelParam	Remove a given parameter from an *instruction* in the given *Pass*
NPF_PokeInst	Change an arbitrary part of an *instruction* in the given *Pass*

Table 2: Selection of `libnpf` APIs

communication mechanism in the `setsockopt()` system call. Similarly, NPF filter programs are attached to the same structure `sk_filter` as the LSF filter programs, with a flag set to tell two kinds of programs apart. Once in action, packets share the same delivery path no matter which packet filter is being used, and only differ in the call to different filtering engines. As a result, our implementation requires minimum modifications to existing kernel codes. It can coexist with the LSF packet filter, and is compatible with the widely used `libpcap` user library.

ii. Userland Library

Being the LSF userland supporting library, `libpcap` provides a well-designed set of powerful routines to control and communicate with the kernel portion of LSF as well as utility functions for tracing and manipulating captured packets. The kernel portion of NPF is compatible to `libpcap`. The user applications of NPF only need a separate mechanism to communicate with the NPF kernel portion, and can continue using the existing `libpcap` routines to receive captured packets. Therefore, instead of hacking `libpcap` to incorporate NPF, we decide to leave it intact and develop a set of complementary libraries.

Filtering criterion				
ip src net 192.168.254.0/24 and tcp dst port 23				
NPF filter				
Pass.Inst(0)→EtherIP_TCP_NonFrag()				
Pass.Inst(1)→Ether_IPSrc(0xFFFFFF00, 0xC0A8FE00)				
Pass.Inst(2)→EtherIP_TCUDPDst(23)				
LSF filter				
(00)	ldh	[12]		
(01)	jeq	#0x800	jt 2	jf 13
(02)	ld	[26]		
(03)	and	#0xffffff00		
(04)	jeq	#0xc0a8fe00	jt 5	jf 13
(05)	ldb	[23]		
(06)	jeq	#0x6	jt 7	jf 13
(07)	ldh	[20]		
(08)	jseq	#0x1fff	jt 13	jf 9
(09)	ldxb	4*([14]&0xf)		
(10)	ldh	[x + 16]		
(11)	jeq	#0x17	jt 12	jf 13
(12)	ret	#96		
(13)	ret	#0		

Table 3: A filtering Criterion and Corresponding
LSF and NPF Filter Programs

As shown in Table 2, the C library `libnpf` provides a set of API functions for the convenient manipulation of NPF filter programs. The compatibility functions to existing LSF filters are also included, making transitions from LSF to NPF effortless. In addition, we also implement a C++ library *oonpf*, providing object oriented filter program control and manipulation, as well as better debugging support. Table 3 shows a common filtering criterion expressed in `pcap` primitives, the NPF filter program using *oonpf* library, and the compiled LSF code as reference. It illustrates the clear logical connection and easy transformation between the NPF filter program and the `pcap` language primitive.

iii. Optimized Implementation

In addition to the NPF architecture, the optimization techniques used in NPF implementation are also essential to the performance improvement of packet filtering. We have made several revisions of our code and gained up to 50% performance speedup in comparison to the initial implementation. Note that these optimizations are only effective in the CISC-like architecture. The major optimizations we employ are listed as follows.

The first optimization is to use bit, instead of byte, operations. For example, in programming the filtering engine program counter, while the natural response is to use an integer, a much faster method of tracking the execution point is to use bit mask.

The second optimization is to hybridize the SIMD indicator with the instruction identifier. It is undesirable to use loops in evaluating SIMD instructions, because (1) the number of loops demanded by the usage context is often very small, which leads to a large percentage of branch miss predictions, and (2) loop counter takes a precious general purpose register, which may otherwise be used for computation acceleration. Our optimization technique greatly speeds up the SIMD instruction execution, because it unrolls the evaluation loop and completely avoids the related performance complications; as well as reduces the SIMD instruction decode time by eliminating extra operations to check the SIMD indicator.

The third optimization handles the duplicated Pass evaluation. As shown in Figure 2, NPF supports hierarchical relationship between Passes to allow for “passive incremental optimization”. Therefore, the filtering engine needs to perform evaluation in a tree structure. Because the recursive evaluations are impractical for in-kernel programs, the only alternative is the iterative approach, which requires simple local storage (a stack or queue). Using indexed array is too slow in that every storage access involves too many operations: accessing, modifying, storing array index, loading array base address, dereferencing and loading element. NPF obtains the fastest storage by utilizing the stack in hardware, with which accessing an element is as fast as executing a "`push`" or "`pop`" instruction. By keeping the frame-pointer for the evaluation engine function, NPF can safely utilize the local stack without interfering with local variable accesses

CHAPTER V.

EVALUATION

In this chapter, we evaluate the performance of NPF in comparison with that of LSF. We also attempt to compare NPF to FFPF [4], the latest packet filter in the literature. However, due to FFPF code maintenance problem, as of now we have not been able to use FPL-3, the latest version of the filter language developed for FFPF.

We compare NPF with LSF on both static and dynamic filtering tasks. The evaluations on static filtering tasks aim at revealing the performance of the filter engine, while the evaluations on dynamic filtering tasks focus on uncovering the overhead of the filter update. We conduct the experiments on four different platforms with different computing architectures. The configurations of these machines (PC1-4) are listed in Table 4 alone with the traffic generator (PCS).

i. Testbed Setup

To evaluate the filtering performance of NPF in a realistic but controllable environment, we set up a test-bed, as shown in Figure 4. A Gbps SMC managed switch is used to provide connections between machines. The switch emulates the gateway through which a local area

Machine	CPU	L2 Cache	Memory	FSB
PC1	1 * Intel Pentium 4 2.8GHz (32-bit)	1MB	768MB	533MHz
PC2	2 * Intel Xeon 2.8GHz (32-bit w/ HyperThreading)	512KB	1GB	800MHz
PC3	1 * Intel Pentium 4 3.6GHz (EM64T w/ HyperThreading)	2MB	1GB	800MHz
PC4	2 * Intel Xeon 2.0GHz (EM64T DualCore)	4MB	4GB	1333MHz
PCS	2 * Intel Xeon 3.06GHz (32-bit w/ HyperThreading)	512KB	2GB	533MHz

Table 4: Testbed Machine Configurations

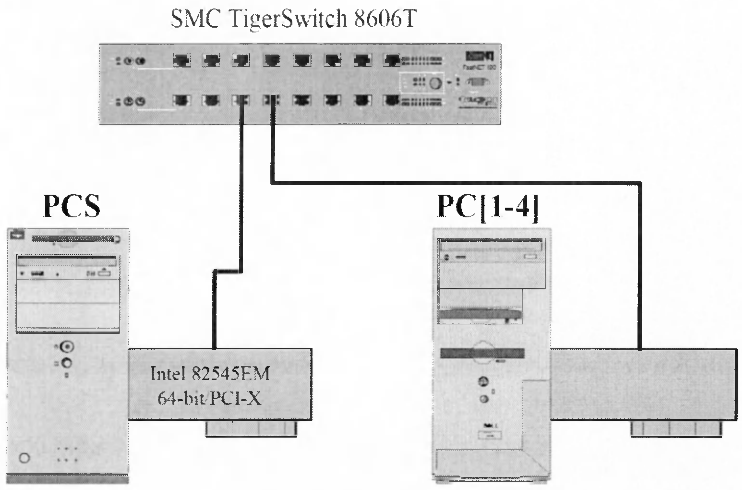


Figure 4: Testbed Setup

network such as an enterprise or campus network connects to the Internet by directing traffic from the specified source port to the mirror port. We use the packet generator machine (PCS), which is connected to the source port, for replaying traces, while one of the four machines (PC1-4) connected to the mirror port receives all the replayed traffic and plays the role of a monitoring device deployed in a real environment.

ii. Static Filtering Performance

Because NPF differs from LSF only in the filtering engine design and implementation, the number of CPU cycles spent inside the filtering engine becomes a natural choice of the performance metric in this study. This metric is gathered by taking the difference of the x86 Time-Stamp Counter (TSC) just before and right after the filtering engine call. Due to the fact that the software interrupt NET RX runs on only one processor and is non-preempted, we can safely use the TSC difference as a speed metric without considering the SMP related complications, such as the fact that TSC is not always synchronized across processors.

Filter	Description	LSF Inst.#	NPF Inst.#
1	"" (Accept all packets)	1	0
2	"ip"	3	1
3	"ip src or dst net 192.168.2.0/24"	10	2
4	"ip src net 192.168.2.0/24 and dst net 10.0.0.0/8"	10	2
5	"ip and tcp port (ssh or http or imap or smtp or pop3 or ftp)"	23	2
6	"ip and (not tcp port (80 or 25 or 143) and not ip host (...)" (The ellipsis mark stands for 38 IP addresses ORed together.)	95	10

Table 5: "Static Filters" Specified in `pcap` Language
and the Corresponding Compiled LSF and NPF Filters' Instruction Count

As shown in Table 5, six different filtering criteria with increasing complexity are used for evaluation. Their instruction numbers in LSF and NPF are also listed for comparison. To show the performance margin that may be exploited, we implement a set of hand-coded optimized C filters, which perform the same filtering functions and serve as the performance ceiling for comparison. We compile these C filters using `gcc` "Os" (optimize for size) flag.

To obtain consistent and reproducible results, we collect a 1.1GB trace containing 14,260,556 packet headers (75 bytes snap length) at the gateway of a local area network. We playback the trace file at the fixed rate of 250,000 (i.e., 250K) packet-per-second (pps) by using `tcpreplay [20]`. Assuming an average of 500 bytes per packet, the playback rate represents a fully utilized 1Gbps link bandwidth. The per-packet CPU cycles consumed by NPF and LSF filtering engines are measured by a short piece of kernel code, which accumulates the CPU cycles (measured by TSC) used for each call to the filtering engine, and averages the value over a given number of packets (say 100,000). We measure the average cycles spent in accepting and rejecting packets separately, and select the larger value of the two as the filter performance data. We prefer the larger number to other measures such as the weighted average, because we find that the worst average is much less affected by network

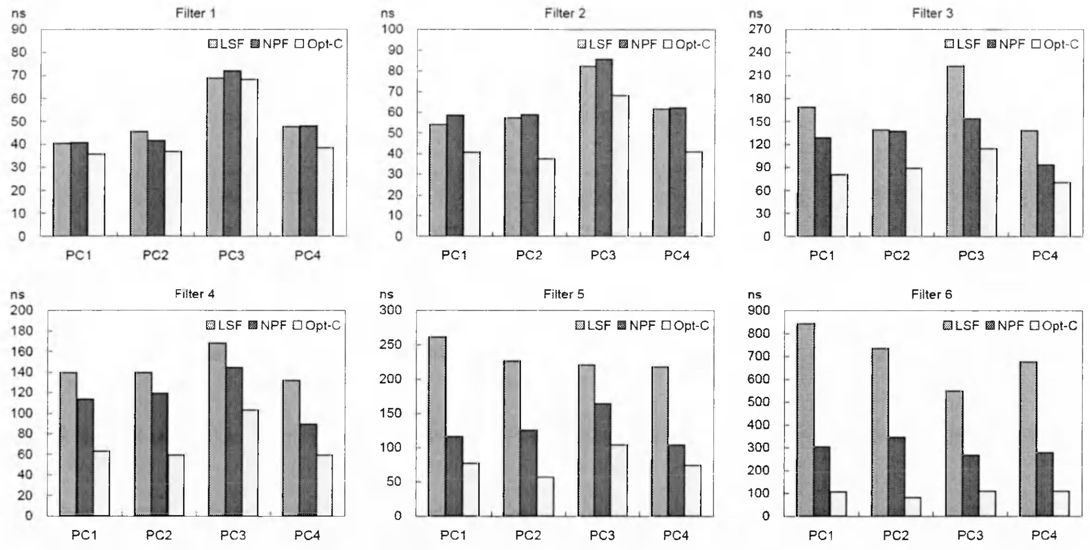


Figure 5: Per-packet Processing Time for Each Filtering Criterion

traffic than any other measures.

Given that processors with different frequencies spend different numbers of clock cycles in executing the same sequence of instructions, we cannot directly compare clock cycles across different machines. Instead, we convert the raw CPU cycles into the actual processing time by dividing the corresponding machine's processor frequency. The converted actual processing times are presented in Figure 5 and Figure 6.

Figure 5 shows the per-packet processing time of LSF and NPF on four machines for each filtering criterion (called "filter" for short in the following), respectively.

Filters 1 and 2 are the simplest criteria. They are designed to show the minimum overhead induced by the filtering engine. The corresponding results show that NPF and LSF run approximately at the same processing speed. Both NPF and LSF have comparable performance to the optimized C filters. These results are expected since NPF inherits the pseudo-machine abstraction as well as the control flow model from LSF.

Filters 3 and 4 are light load criteria designed to demonstrate the filtering engine's capability for basic packet classification. The corresponding results indicate that NPF has a moderate performance advantage over LSF on all machines. In filter 3, speedups are from 1.3% to 46.8% with an average of 31.0%. In filter 4, NPF outperforms LSF on all the four machines; speedups are from 16.0% to 47.5% with an average of 25.8%. Compared to the optimized C filters, LSF on average runs 88.9% slower for filter 3 and 110.9% slower for filter 4; while NPF on average runs 45.2% slower for filter 3 and 68.5% slower for filter 4. The major performance gain of NPF is attributed to its CISC-like instruction set. With much smaller number of instructions in the filtering program, NPF reduces instruction interpretation time and achieves higher performance than LSF.

Filter 5 is a moderate load criterion designed to test the filtering engine's capability of handling a highly specific filter criterion. The corresponding results show that NPF outperforms LSF by a significant amount on all machines. Numeric data reveals that NPF's speedup ranges from 34.2% to 126.0%, with an average of 87.6%. Compared to the optimized C filter, LSF on average runs 210.9% slower, while NPF on average runs 67.3% slower. Again, the significant speedup of NPF is due to its architectural advantages, specifically, SIMD instructions. The ability to pack many operands (12 for TCP/UDP ports) in one instruction and batch the execution of comparison operations within a single filter engine "cycle" enables many-fold reduction in the cost of instruction interpretation, and brings the performance of

NPF close to that of the optimized C filter.

Filter 6 is a heavy load, “real life” criterion, transformed from a `pcap` filter supplied by our system administrator. In order to express this filtering criterion, the construction of the NPF filter program uses the optimization features we implemented, including SIMD instructions for reducing instruction interpretation cost, and *Pass* duplication for eliminating redundant instruction evaluations. This filter is used as an ultimate performance test of NPF in a realistically heavily loaded environment. The corresponding results show even higher performance speedup of NPF against LSF. Numeric data shows that NPF’s speedup ranges from 104.7% to 176.5%, with an average of 134.3%. The over 100% speedup clearly demonstrates that utilizing both CISC-like ISA and SIMD with proper optimization, NPF can significantly increase the filtering efficiency by better unleashing the power of the underlying hardware. Comparing to optimized C filter, LSF on average runs 602.2% slower, while NPF on average runs 201.9% slower: NPF performs much closer to the optimized C filter than LSF.

Figure 6 presents a cross comparison of filter execution time for LSF, NPF, and optimized C filters among all machines categorized by six filtering criteria. It provides a good overview of the static filtering performance. When the filtering criteria are simple and light load, LSF, NPF and optimized C filters have nearly indistinguishable performances. As the criteria become more complex and heavy load, the differences of filter execution time among the three filtering engines grow. Although both NPF and LSF run slower than the optimized C filters, the filter execution time of NPF grows at a much slower rate than that of LSF, and thus

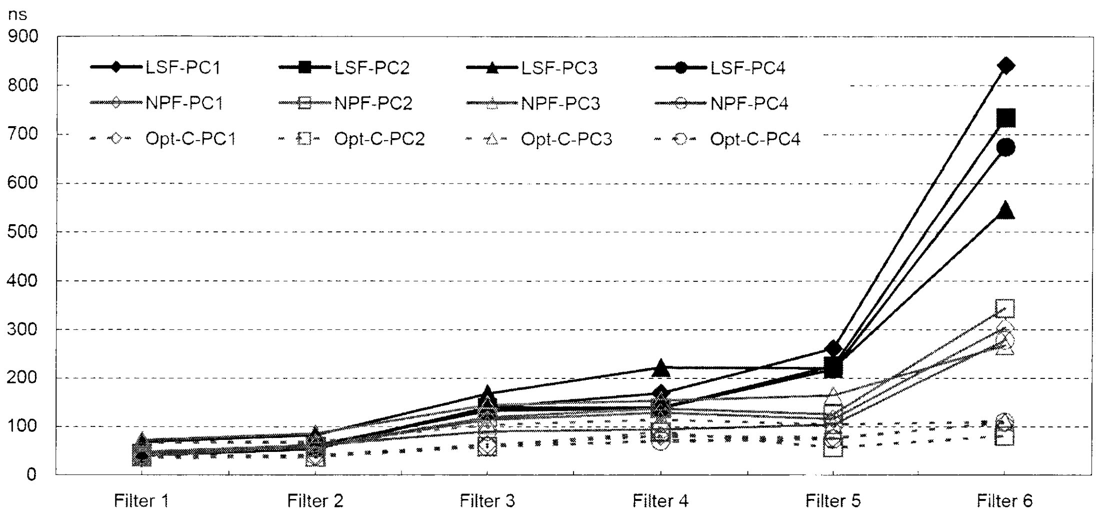


Figure 6: Per-packet Processing Time Comparison for All Filtering Criteria

NPF achieves much closer performances to the optimized C filters than LSF.

iii. Dynamic Filtering Performance

Since capturing FTP passive mode traffic is a typical dynamic filtering task, we apply NPF and LSF to filter FTP traffic in passive mode and compare their performance, in terms of the total number of captured connections/packets and the filter update latency. We use a generated FTP traffic trace for the evaluation of dynamic filtering. The trace consists of up to 25 concurrent FTP sessions for passive mode data transfers, all with different source and destination IP addresses and port numbers. The number of concurrent sessions increases gradually from 0 to 25 at the beginning and then decreases back to 0 at the end. We replay the trace at the rates of 100K, 250K, 333K and 555K pps (the highest sending rate PCS can sustain), respectively, using `tcpreplay` on PCS and perform filtering tasks through NPF and LSF separately on PC1 to PC4.

For experiments using LSF, we employ "`(ip and tcp port ftp)`" as the initial filtering criteria to capture FTP control packets. When the set-up of each FTP data connection is detected (by analyzing the captured FTP control packets for new data connection information), we append the filtering criterion "`or (ip xx and tcp port yy)`" to the existing ones. The "`ip xx`" and "`tcp port yy`" refer to the FTP server's IP address and data port number, respectively. When the teardown of a data connection is found (by looking for a TCP FIN control packet), we remove the corresponding filtering criterion. For experiments using NPF, we initialize the first *Pass* corresponding to the filtering criterion "`(ip and tcp port ftp)`", in order to capture FTP control packets. As concurrent FTP data connection set-up and tear-down events are detected, we create and remove *Passes* corresponding to the filtering criteria "`(ip xx and tcp port yy)`", respectively. We apply "passive incremental optimization" by using function `NPF_NewPass()` only for the first data session and function `NPF_DupPass()` for the rest of data sessions.

We collect two types of performance metrics, the total number of captured connections/packets and the filter update latency. The former is measured using a simple counter, and the latter is derived from the values of the CPU Time-Stamp Counter before and after invoking function calls for filter update. Each experiment is repeated 10 times, and the mean and standard deviation are calculated. As mentioned before, NPF and LSF share the same Linux packet filter framework, including packet sniffing mechanism, traversal path in kernel, and user-kernel interface. Except for the filter engine, all other conditions such as the in-kernel NAPI [18] mmap ring size are the same. Thus, fairness is ensured in our

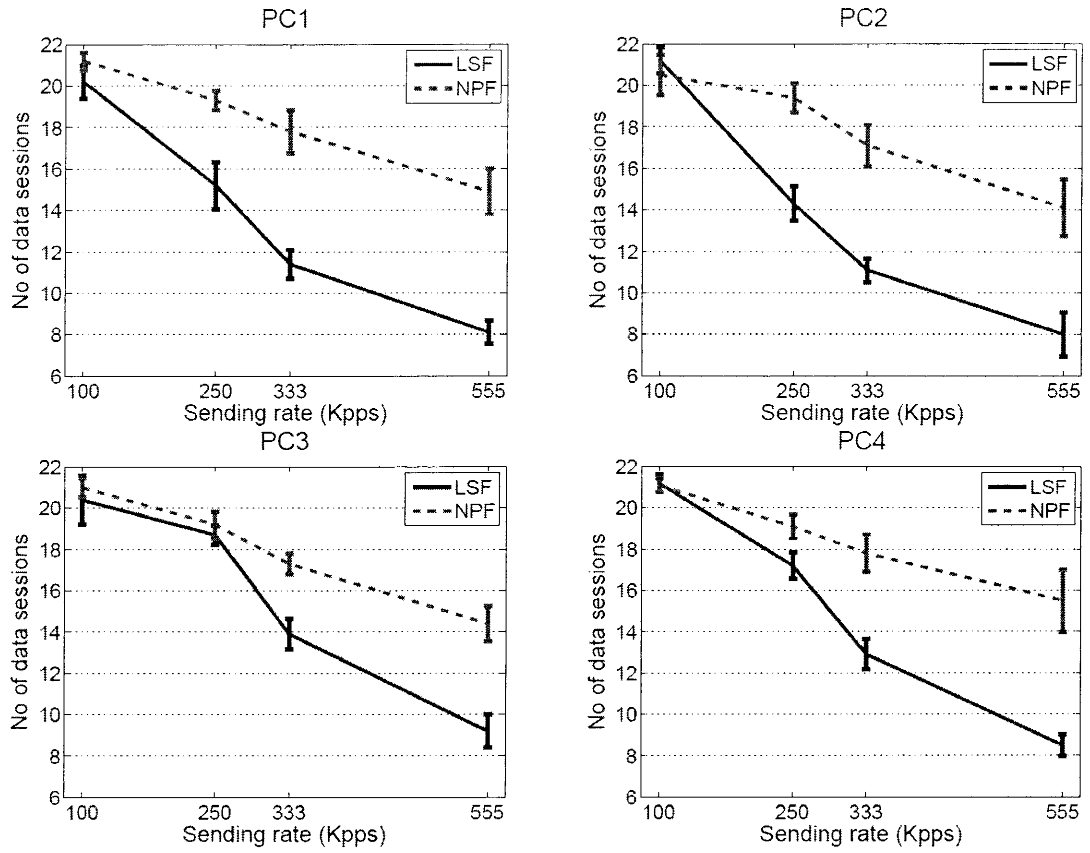


Figure 7: Captured FTP Data Connections at Different Traffic Rates

experiments for performance comparison between NPF and LSF.

Figure 7 illustrates the comparison between LSF and NPF on the number of captured FTP data connections. The blue solid curves are for LSF and the red dotted curves are for NPF. The vertical bars represent the standard deviations. Neither LSF nor NPF can capture all data session at any rate, i.e., both have packet drops. It is clear that NPF has higher capability of capturing data sessions than LSF. The trend of curves in all figures suggests that the higher traffic rates are, the larger difference in captured data sessions between LSF and NPF will be. Although both LSF and NPF see almost the same amount of data sessions at the lowest rate,

	100K pps		250K pps		333K pps		555K pps	
	LSF	NPF	LSF	NPF	LSF	NPF	LSF	NPF
PC1	924 + 256K	948 + 270K	642 + 153K	848 + 231K	519 + 95K	727 + 204K	364 + 39K	649 + 151K
PC2	946 + 267K	933 + 262K	636 + 141K	829 + 233K	520 + 88K	719 + 190K	382 + 45K	584 + 113K
PC3	918 + 258K	956 + 269K	704 + 167K	820 + 229K	560 + 133K	758 + 199K	418 + 56K	611 + 145K
PC4	946 + 269K	940 + 270K	690 + 170K	829 + 226K	574 + 99K	754 + 202K	370 + 42K	665 + 153K

Table 6: Captured FTP Packets at Different Traffic Rates

NPF catches 60% to 90% more sessions than LSF at the highest rate. Note that the capture of a data session can be initiated only when the corresponding set-up control packet is detected.

Table 6 lists the average numbers of the FTP packets captured by LSF and NPF at different traffic rates. The format is "control packet number + data packet number". The data manifest that in general the performance gap between NPF and LSF on packet capture matches that on session capture, but to a larger degree. Particularly, the gain of NPF over LSF on data packet capture exceeds that on control packet capture, although both grow with increasing rates. This is attributed to the fact that the criterion for capturing FTP control packets is placed in the foremost of the filter, which results in the constant comparison cost for capturing control packets. In contrast, comparisons for data packets need to traverse half of

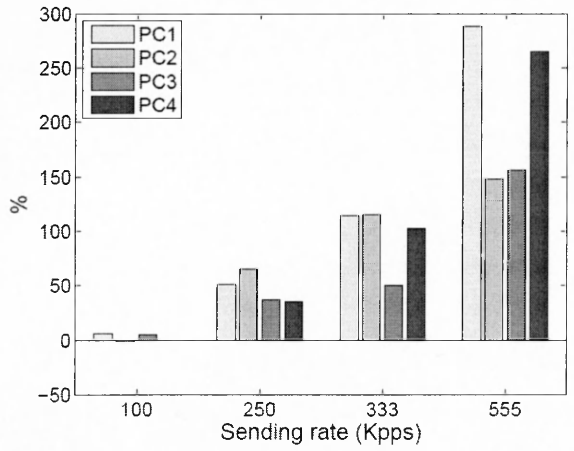


Figure 8: Performance Gain of NPF over LSF on the Number of Captured FTP Data Packets

the whole filter on average. Due to LSF's higher interpretation cost, the performance of LSF in capturing FTP data packets degrades much faster than that of NPF, with increasing length of filtering criteria. Figure 8 shows the performance gain of NPF against LSF on the number of captured FTP packets. At 100K pps, NPF performs equally well as LSF. At 555K pps, however, NPF captures 148% to 288% more packets than LSF.

The filter update latencies for LSF and NPF on PC2 are shown in Figure 9 (a) and (b), respectively, in which the most significant difference lies in the order of magnitude of the y-axis unit. While updating an LSF filter takes $\times 10^7$ ns, updating an NPF filter takes only $\times 10^4$ ns. By eliminating filter compilation and security checking, NPF gains three orders of magnitude speedup against LSF. Another noticeable feature here is the growing latency in LSF filter updates against "saw tooth" latency in NPF filter updates, under the context of increasing concurrent sessions.

As shown in Figure 9 (a), both concurrent session number and traffic rate affect the filter update latency of LSF. When the number of concurrent sessions increases, the filtering criteria expressed in `pcap` language become longer. Accordingly, the compilation procedure consumes more CPU cycles. The cost of security checking also increases as the filter program size grows. With the increase of traffic rates, more CPU cycles are taken by the filtering engine and less can be spared for filter replacement compilation, resulting in longer filter update latency. In contrast, the filter update latency of NPF is overall insusceptible to the change of traffic rates and concurrent sessions. Since we measure latency at the application

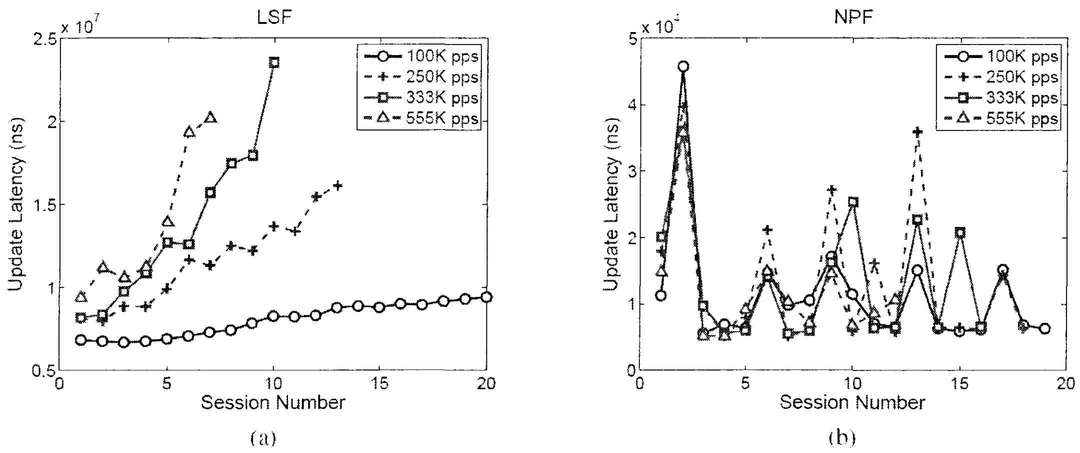
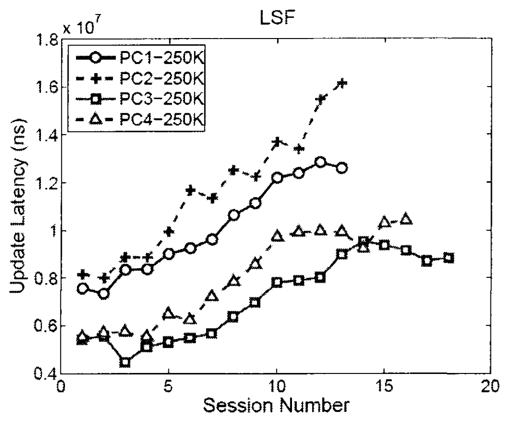


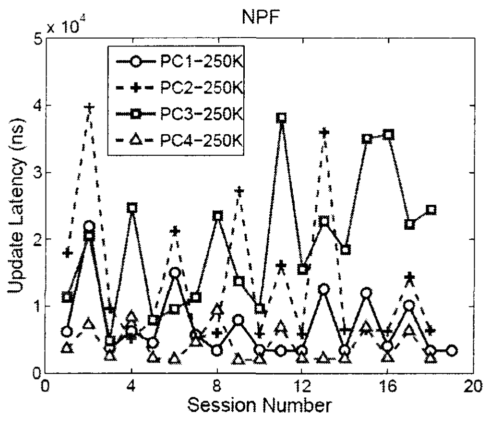
Figure 9: Average Filter Update Latency at Different Traffic Rates on PC2

level and the overall update latency of NPF is very small, the overhead caused by indeterministic factors such as context switch and process scheduling is not constant and produces the spikes in Figure 9 (b). Without filter compilation and security checking, NPF performs filter update in an incremental manner. Adding or removing a concurrent session takes almost constant time. Moreover, because the procedure of updating an NPF filter is quite simple and the overhead of update is mainly caused by system calls, increasing traffic filtering load has very limited (almost negligible) impact on its latency. This further explains why NPF achieves much higher performance than LSF at high traffic rates.

Since all machines exhibit similar behaviors in filter update at different traffic rates, we present a cross machine comparison on filter update latency only at the traffic rate of 250K pps in Figure 10. Similar to Figure 9, the filter update latencies of LSF are three orders of magnitude longer than those of NPF, and (approximately linearly) increase with the addition of more concurrent sessions. By contrast, the dynamics of NPF filter update latency are



(a)



(b)

Figure 10: Average Filter Update Latency on Different Machine at 250K pps

independent of the number of concurrent data sessions on any machine.

CHAPTER VI.

CONCLUSION

This thesis presents the design and implementation of a New Packet Filter. NPF provides an elegant, fast, and efficient packet filtering approach to handle the upraising challenge of high speed network monitoring with dynamic filter updates. The key features of NPF lie in its low filter update latency and high execution efficiency. NPF attains these performance advantages by embracing several major design renovations: (1) the highly specialized CISC-like instruction set eliminates the filter criteria compilation, resulting in high filter execution efficiency; (2) the DFA computational model removes the necessity of security checking and significantly reduces the filter update latency; and (3) the SIMD technology further boosts the filter execution efficiency.

Our extensive experiments have validated the efficacy of NPF and demonstrated the superiority of NPF against the *de facto* packet filter, BPF. For static filtering tasks, NPF runs as fast as BPF on simple filtering criteria, but is up to three times as fast as BPF on complex filtering criteria. We also used optimized C filters as the performance ceiling, and found that NPF performs much closer to the optimized C filters than BPF. For dynamic filtering tasks, the filter update latency of NPF is three orders of magnitude shorter than that of BPF, and NPF can capture up to 288% more packets than BPF.

There are many avenues we would like to further experiment and exploit in NPF. For instance, we will explore the multi-thread expansion of NPF, and develop a hardware optimized filter engine. We will make use of the extra registers supplied in the x86 64 processors for further performance improvement. Moreover, we envision that x86 high performance multimedia instructions (such as MMX and SSE) can also be used to accelerate the packet processing.

REFERENCES

[1] J. Apisdorf, k claffy, K. Thompson, and R. Wilder. OC3MON: Flexible, affordable, high performance statistics collection. In Proceedings of USENIX LISA'96, pages 97–112, Chicago, IL, September 1996.

[2] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar. PathFinder: A pattern-based packet classifier. In *Proceedings of USENIX OSDI'94*, pages 115–123, Monterey, CA, November 1994.

[3] A. Begel, S. McCanne, and S. L. Graham. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *Proceedings of ACM SIGCOMM'99*, pages 123–134, Cambridge, MA, August 1999.

[4] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis. FFPF: Fairly fast packet filters. In *Proceedings of USENIX OSDI'04*, pages 347–363, San Francisco, December 2004.

[5] J. Cleary, S. Donnelly, I. Graham, A. McGregor, and M. Pearson. Design principles for accurate passive measurement. In *Proceedings of PAM'00*, pages 1–8, Hamilton, New Zealand, April 2000.

[6] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational experiences with high-volume network intrusion detection. In *Proceedings of ACM CCS'04*, pages 2–11,

Washington, DC, October 2004.

[7] D. R. Engler and M. F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of ACM SIGCOMM'96*, pages 53–59, Stanford, CA, August 1996.

[8] G. Insolvibile. Inside the linux packet filter. http://www.linuxjournal.com/article/4852. Linux Journal.

[9] G. Insolvibile. Inside the linux packet filter, part ii. http://www.linuxjournal.com/article/5617. Linux Journal.

[10] S. Ioannidis, K. G. Anagnostakis, J. Ioannidis, and A. D. Keromytis. xPF: Packet filtering for low-cost network monitoring. In *Proceedings of IEEE HPSR'02*, pages 121–126, Kobe, Japan, May 2002.

[11] V. Jacobson, C. Leres, and S. McCanne. Tcpdump(1). *Unix Manual Page*, 1990.

[12] G. R. Malan and F. Jahanian. An extensible probe architecture for network protocol performance measurement. In *Proceedings of ACM SIGCOMM'98*, pages 215–227, Vancouver, Canada, September 1998.

[13] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 Winter USENIX Technical Conference*, pages 259–269, San Diego, CA, January 1993.

[14] S. McCanne, C. Leres, and V. Jacobson. Libpcap. availble at http://www.tcpdump.org/. Lawrence Berkeley Laboratory, Berkeley, CA.

[15] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the 11th ACM SOSP*, pages 39–51, Austin, TX, November 1987.

[16] A. Moore, J. Hall, C. Kreibich, E. Harris, and I. Pratt. Architecture of a network monitor. In *Proceedings of PAM'03*, La Jolla, CA, April 2003.

[17] U. of Waikato. The DAG project. available at http://dag.cs.waikato.ac.nz/.

[18] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond softnet. In *Proceedings of USENIX 5th Annual Linux Showcase and Conference*, pages 165–172, Oakland, CA, November 2001.

[19] SCAMPI. A scalable monitoring platform for the internet. *Leiden University (in collaboration)*, March 2002.

[20] A. Turner. Tcpreplay. http://tcpreplay.synfin.net/trac/.

[21] J. van der Merwe, R. Caceres, Y. hua Chu, and C. Sreenan. mmdump – a tool for monitoring internet multimedia traffic. *ACM Computer Communication Review*, 30(4), October 2000.

[22] G. Varghese. Network algorithmics - an interdisciplinary approach to designing fast

networked devices. In *Morgan Kaufmann Publishers*, 2005.

[23] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Proceedings of the 1994 Winter USENIX Technical Conference*, pages 153–165, San Francisco, CA, January 1994.

VITA

Zhenyu Wu

Zhenyu Wu was born in Chengdu, Sichuan, China on August 03, 1982. He graduated from High School of Sichuan Normal University in Chengdu, Sichuan, China in June 2000. Zhenyu Wu received his B.S. at Denison University in May 2005, with double degrees in Computer Science and Physics.

In August 2005, the author entered the College of William and Mary as a graduate student in the Computer Science Department. Zhenyu Wu will defend his thesis in April 2007. He will continue studying in the Computer Science Department for his Ph.D. degree.