PFC: A New High-Performance Packet Filter Cache

Chuan-Hsing Shen, * **Tein-Yaw Chung**
Department of Computer Science and Engineering Yuan Ze University
* csdchung@saturn.yzu.edu.tw

Abstract-*As communication technology advances, network capacity grows exponentially in recent years. The performance of network monitoring tools is getting more critical as they must process much lager number of packets in a unit of time than ever before. A common core component in any network monitoring tools is a packet filter which processes every packet header and passes those packets matching some filter rules to user spaces for further processing. In this paper, a packet filter architecture called Packet Filter Cache (PFC) is proposed to improve the performance of existing packet filters. The PFC architecture adds a filter rule cache before an existing packet filter. Instead of caching instruction set as in Warm cache, the filter rule cache stores the hash value of a filter rule as a hash table entry that can be searched in one memory access. By taking advantage of the hash lookup speed, PFC can boost filtering performance by using only small cache size. Moreover, PFC also caches unmatched packet flows to achieve high hit rate. Since PFC is only a cache mechanism added before a traditional packet filter, it does not need to re-engineer existing filter module and hence can be applied on most packet filters. Simulation shows PFC can improve the processing time about four times at cache hit rate of 70%.*

Keyword: cache, packet filter, packet classification, un-matched flow.

1. Introduction

The ever-increasing complexity in network infrastructures is making critical demand for network monitoring tools. Network monitoring tools allows individual user processes to have great flexibility in selecting which packets they will receive. A common core component of network monitoring tools is a packet filter [1] which is a programmable selection criterion for selecting packets from a packet stream. For the majority of networks, such functions are implemented using commodity components: PC workstations or servers running free operating systems and open-source monitoring tools like EtherReal[2], Tcpdump[3], NeTraMet[4], ntop[5], and snort[6]. Deploying packet filter as a kernel agent can minimize the packet copy across the kernel/user-space protection boundary when monitoring [1]. Currently, most of monitoring tools rely on Berkeley Packet Filter (BPF) facility [2], which allows them to capture packets from the network interface.

As the speed of network links continues to increase, the use of commodity components and BPF is becoming inefficient. Over the past few years a considerable number of studies have been made on packet filter and packet classification. Previous work on packet filters make an effort to investigate flexible and extensible filter abstractions but sacrifice performance[7-9], or focus on low-level, optimized filtering representations but sacrifice flexibility[10-12]. They have proposed solutions [13,14] for some particular situation, but are not general enough to handle all types of filters. Furthermore, the aforementioned works [10-14] require significant effort in re-engineering the existing body of BPF.

In this study we attempt to provide high performance network monitoring with minimal changes to existing infrastructure. To make this possible, we enhance BPF by adding a packet filter cache (PFC) before BPF. Although a cache mechanism called warm cache has been proposed before, it is mainly used to cache filter instructions to reduce packet processing time. However, it only achieves little performance improvement and thus is rarely used. PFC, on the other hand, is a processing filter rule cache but not an instruction cache. When cache hit ratio is high, most packets are processed at the packet filter cache without going through a packet filter. Therefore, packet processing time is significantly reduced. To improve cache hit rate, we also cache unmatched packet flows to prevent some packets always falling through all the filters. Simulation results show that with PFC, the resulting system can achieve high performance and low system overheads. At the same time, PFC can retain the simplicity, portability and compatibility with existing tools and the appealing maturity and stability of existing infrastructure.

2. Packet Filter Cache

This section introduces the design principle of PFC and its architecture and operation. The first section overviews the design concept of PFC and then introduce the PFC architecture. Next, the organization of cache tables and how a cache table is generated is

This research was supported by the National Science Council, Taipei, Taiwan, R.O.C., Project no. NSC92-2213-E-155-037

described. Following that, step by step packet processing through PFC is illustrated to show how PFC works.

2.1. Architecture

The packet filter cache (PFC) uses two novel mechanisms, filter rule caching and unmatched flow caching. Traditional warm cache saves process instructions to speed up packet processing. However, it requires large cache size to effectively improve packet processing speed and has low cache hit rate. Instead of caching instruction set, PFC cache hashed filter rules in PFC to speed up filter rule search as compared to traditional linear search of the warm cache. Since a hashed filter rule uses only a small cache size, even using a small size cache, a large number of filter rules can be cached. Thus, caching hashed filter rules can increase hit rate significantly.

In order to make hashed filter rule caching possible, PFC maps filter rules into a number of hash tables, each with a distinct mask that is used to derive prefixes from packet header fields for filter rule check. The hashed filter rule is saved in a cache table whose mask matches with that of the cache table. The search for a filter rule in PFC can then be efficiently done by simple hashing and comparison.

Suppose we have a filter database with N filters, these filters are mapped to m distinct masks. Since m tends to be much smaller than N in practice, search linearly through the mask set is likely to be much faster than the linear search through the database. However, using cache tables to replace full packet filter rules can make the number of cache table very large, in the worst case up to $O(W^d)$, where W is the number of possible entries for a field and d is the number of fields in a filter rule. Therefore, in PFC, a prefix expansion approach is adopted to reduce the number of cache tables. This is to be introduced later.

In PFC, unmatched packet flows are also cached. A packet that un-matches any filter rule will fall through all filters and cause heavy processing load. For example, if a network monitor filters out 10% of packet streams for analysis, the other 90% of packets will fall through all filters and make the packet filter experience heavy load. By caching unmatched flows, PFC can achieve much better cache hit rate and significantly reduce filter processing load.

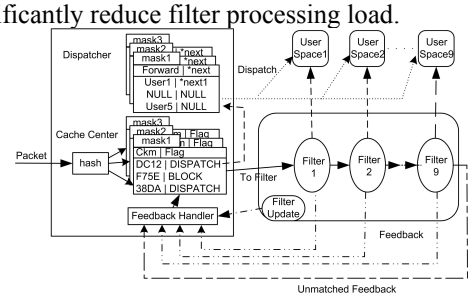

Fig. 1: PFC Architecture

The PFC architecture includes four components: traditional packet filter, cache center, cache dispatcher, and feedback handler as shown in Fig. 1. The traditional packet filter in principle can be any existing packet filters. In the paper, without loss of generosity, we use BPF as our filter engine. BPF is one of the most popular packet filter engine and used in most BSD systems. The Cache Center includes a hash function and cache tables. When a packet arrives, the Cache Center will do hash function for the packet and determine where the packet should go. The Cache Dispatcher forwards a packet to each of its matched user space. The Feedback Handler receives filter rule feedbacks from the packet filter and then writes the filter rules to the cache tables in the Cache Center and the Cache Dispatcher. Filter update creates or removes cache tables from the Cache Center when filter rules are inserted or deleted. As can be seen from Fig. 2, PFC does not need to re-engineer the body of existing BPF. What needs to be modified to the existing BPF is to create feedback links and to connect them to the PFC Feedback Handler. Therefore, PFC can be applied easily to any existing packet filter architectures. The following sections offer further detail on PFC.

2.2. Cache Table Generation and Maintenance

In PFC, each cache table is associated with a mask and each cache entry in a cache table is an entry of (hash, checksum, flag, dispatch). The hash value is a hashing of concatenated prefix value derived from each field of a filter rule. The algorithm for prefix hashing is illustrated in Fig. 2. The hash is used to map a filter rule into a cache table. The multi-dimensional nature of filter rule search operation is removed by combining several fields into one search key and treating the problem as single-field search. We use flag and dispatch field to achieve the demand of multi dispatch. The flag field is either *DISPATCH* or *BLOCK* type. It indicates if a matched packet should be forwarded to a user space or be blocked. If the flag field is *DISPATCH*, the packet will be forwarded to user spaces; otherwise, it will be blocked.

Build HashedFilterRule: /* called on filter rule hashing */
for i = 1 to K /* K is number of filter fields */
M*i* := *MaskLookup*($P[i]$);
C := $M_1M_2…M_K$;
H := *Hash*(C)
Return H;

Fig.2: Pseudo Code for Hash Cache Generation

Let's take Table 1 as an example. According to the prefix of each field in the filter rules, a mask set can be generated as shown in Table 2. For instance, [16, 8, 0, 8] is a 4-dimensional tuple that represents a mask corresponding to rule R4 in Table 2, each mask field corresponding to the number of prefix bits of IP source, IP destination, source port, and destination port.

Table 1: Example of filter rule table

Rule	Src Addr	Dst Addr	Src Port	Dst Port	Action
R1	140.138. *	140.*		Eq ftp	User1
R2	140.138. 144.*	140.*		Eq ftp	User2
R3	140.138.	140.*		Eq	User2

	145.*			www	
R4	140.138. *	140.*		Lt 1023	User3
R5	140.138. *	140.*	Eq ftp		User4

All filters having the same mask are mapped to a particular cache table as shown in Fig 3, i.e., these rules require the same number of bits in the IP source, destination fields and so on for filter rule check. A filter rule is then represented by hashing the concatenated prefixes of each field of the filter rule. For example, R4 in Table 1 is represented by the hashed value of the concatenation of 140.138, 140, 0, and 1024.

Table 2: Example of packet filter cache mask table

Rule	Mask	Action
R1	16,8,0,16	User1
R2	24,8,0,16	User2
R3	24,8,0,16	User2
R4	16,8,0,8	User3
R5	16,8,16,0	User4

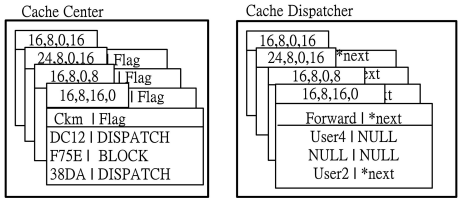

Fig. 3: Example of cache center and cache dispatcher

Complex filter rules require large numbers of hash tables and cause heavy hash table search overhead. The lookup performance of PFC can be improved by reducing the number of distinct mask or number of cache tables via further use of Controlled Prefix Expansion (CPE)[20]. CPE transforms a set of prefixes into an equivalent set of prefixes with longer length and is used to construct multi-bit tries. We expand filter mask length to reduce the number of cache tables whenever possible. An example of filter expansion with one dimension is shown in Table 3, where the prefix of filter is expanded from 01* to prefixes 010* and 011*. After expanding prefix, we get a set of new filter prefix 010* and 011*, which is equal to the original filter prefix 01* and the mask of R5 is no longer needed and hence the respective cache table.

Since hash function is not perfect, prefix of different filter rules may have the same hash value. To avoid hash conflict, PFC uses a secondary hash table named checksum to double check potential hash collision. By using double hash values as the signature of a filter rule, the probability of un-caught hash collision among filter rules can be significantly diminished. Checksum is a hash value from the concatenated value of prefix of each field in a filter. PFC computes checksum by first taking XOR of the prefix of each field in a filter rule. Then, a CRC hash is applied on the XORed value to generate its checksum.

Table 3(A): Before Mask expansion with CPE

Rule	Mask	Action
R1	000*	User Space 1
R2	001*	User Space 1
R3	100*	User Space 2
R4	111*	User Space 3

Rule	Mask	Action
R1	01*	User Space 4

Table 3(B): After mask expansion and prefix expansion

Rule	Mask	Action
R1	000*	User Space 1
R2	001*	User Space 1
R3	100*	User Space 2
R4	111*	User Space 3
R5	010*	User Space 4
R6	011*	User Space 4

The design of hash lookup and checksum verification also can achieve Least Recently Used (LRU) effect. When hash collision happens, the checksum comparison is false and the Feedback Handler will replace the existing cache entry with new one. Although we can extend the cache size by link-list, PFC adopts "replace when collide" strategy. This strategy helps reducing overhead for cache rules searching and updating. With such strategy, a filter rule is replaced when it collides with a new filter rule that matches with a new coming packet, and thus the effect of LRU is achieved. Therefore, a flow with higher traffic rate will hit a cache entry with higher probability and be processed in higher speed.

2.3 Packet Processing

When a packet arrives, it will be first processed in PFC. The header fields of the incoming packet are extracted using each mask of cache tables and hashed. The hash value generated for each cache table is then used to find if an entry with the same hash value exists in the respective cache table. If a hash value is mapped to a not null entry in a cache table, the Cache Center will do checksum verification. When checksum comparison is true, a filter rule match occurs, and an action is taken according to the flag of the entry. On the other hand, if match does not occur, the packet is forwarded to the BPF engine for processing.

During the packet processing in BPF, each matched filter rule is sent to the Feedback Handler. If the packet does not match any filter rule, the feedback function randomly chooses a mask in the Cache Center and generates a new filter rule as feedback. Fig. 4 shows the packet processing flow chart in PFC.

When more than one filter rule has the same hash and checksum, a packet matching these filter rules may be forwarded to more than one user space. We implement multi-forward function in the Cache Dispatcher. The multi-forward function is supported by adding a linking list field in the dispatcher table as shown in Fig. 3. The Cache Dispatcher contains *(Forward, *next)* fields, where *Forward* records the user space information and **next* provides a link to next field that records another user space. When the Cache Dispatcher gets a hash key, it calls the packet

filter forwarding function to dispatch the packet to the destined user space. And then check the **next* field. If the **next* field is not NULL, it does next dispatch, and so on.

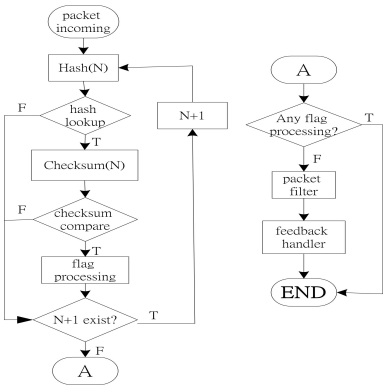

Fig. 4: PFC flowchart

Feedback functions report matched filter rule to the Feedback Handler. It sends messages with a tuple of (mask, hash, checksum, user space, flag). User space field may have two types, one is real user space for matched filter rule, the other is NULL for unmatched filter rule generated by feedback function. The Feedback Handler analyzes the messages and updates the respective cache entry in the Cache Center. The processing of the Feedback Handler is shown in Fig. 5.

FeedbackHandler: /* called on receiving feedback F*/
/* F(*mask, hash, checksum, user space, flag*) */
  Find cache table with *mask*
  Search *hash* in hash table entry
  Compare the *checksum*
  if checksum is the same then
    Add new *user space* information
  else
    Replace *checksum* and *user space* and *flag* information

Fig. 5: Pseudo Code for Feedback Handler

To illustrate how PFC works, we use Fig. 6 as an example. Here, we assume there are three cache tables in the Cache Center. In this example, a packet matches a packet filter and causes collision at checksum value in cache table #1. The packet is then forwarded to BPF for processing. After the Feedback Handler receives a new feedback with new checksum and flag from the feedback function, it will update the cache entry with a new checksum and flag. After that, the remainder of this flow will be forwarded to user space directly by PFC.

Packets that unmatched any rule will generate unmatched hash that is assigned *BLOCK* in its flag field. Blocking these packets can prevent some deny of service attacks and avoid unmatched packets falling through all filter rules. The blocked function is not used in a traditional packet filter but need for firewall and RSVP services. When an un-matched packet goes through the Cache Center for the first time, it will be passed to BPF because it miss hits in PFC. While it falls through BPF, it triggers an unmatched feedback sent back to the Feedback Handler, which randomly selects a cache table and creates an entry with block flag. When PFC receives another packet in the same flow as previous one, the packet will be blocked in PFC and not passed to BPF.

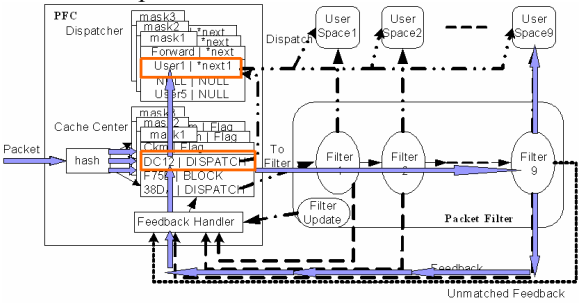

Fig. 6: Matched Cache in Cache Center

3. Performance Analysis

This section analyzes the performance of PFC using computer simulation. We study the performance of PFC under different traffic load, flow interval distribution and filter matching ratio. The performance of PFC focused in our study is on how large a PFC cache required to perform better, the relation between cache hit rate and cache size, and the impact of traffic interval distribution. The chapter is organized as follows: sub-section 1 describes the simulation approach in our study. Sub-section 2 explains our empirical study on the performance of PFC. Sub-section 3 presents the performance of PFC over different cache sizes, flow interval distribution and hit rate. Sub-section 4 studies the impact of hash collision on the performance of PFC. Finally, in sub-section 5 we translate the impact of cache hit ratio into processing overhead and study the performance of PFC in processing load reduction.

3.1 Simulation Approach

We implement a simulation environment for PFC, including a set of 1000 filter rules and a FCFS queue for packet processing. Each traffic flow has an average traffic interval dynamically generated according to a probability distribution. To simplify simulation, we divide packet arrival time as fixed time slots. Packets of a flow arrive at the FCFS queue following Poisson process except stated otherwise.

To obtain more accurate simulation results, we make 100 packets go through 1000 BPF filter rules and assume the hit rate is constant value. We assume BPF use bit operation and default length of CFG lookup algorithm. PFC uses XOR hash function and 20 hash tables.

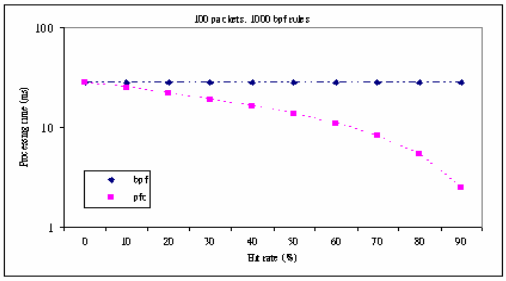

Fig.7: Instruction Simulation PFC Versus BPF

We measure the processing time of BPF with different PFC hit rates. The result is shown in Fig. 7 which will be later used in our simulation analysis. In our experiment, the measurements were made using i386 processors running FreeBSD 4.9-Stable, using a 100Mbit/sec Ethernet. The testing machine has 1816 MHz processor and 128 MB RAM.

3.2. Preliminary Observation

In PFC, a packet is first processed to see if its respective flow of packets has been processed before. If yes, it is dispatched right away using the PFC logic; otherwise, it is forwarded to BPF for further processing. Thus, the average processing time Ttotal for a packet in PFC is as following:

$$Ttotal = Rhit*Thash + (1-Rhit)*(Thash+Tfilter) \quad \ldots (1)$$

Where *Rhit* is the average percentage of packets hit in cache, *Thash* is the average hash function execution time in PFC, and *Tfilter* is the average packet filter processing time through BPF. The performance improvement in processing time reduction can then be expressed as follows:

$$\frac{T_{filter}}{R_{hit} * T_{hash} + (1 - R_{hit}) * (T_{hash} + T_{filter})} \quad \ldots (2)$$

Obviously, when *Thash* is much smaller than *Tfilter*, the processing overhead reduction of PFC over existing PBF becomes:

$$\frac{1}{1 - R_{hit}} \quad \ldots (3)$$

The above preliminary observation assumes infinite cache record size and that *Thash* is much smaller than *Tfilter*. By (3) we see that at hit rate of 75%, PFC architecture outperforms the conventional BPF architecture by four times improvement in processing time.

3.3. Impact of System Parameters

In the simulation experiment, different traffic patterns such as constant distribution, uniform distribution, pareto distribution and exponential distribution are implemented. We first make different number of flows go through PFC with different buffer sizes. Each flow has a constant arrival interval of 10. From Fig. 8, we see the hit rate decreasing drastically when flow size is doubled at buffer size of 8192 byte. The impact of flow numbers on hit rate reduces when buffer size increases. This is obvious because the larger is the buffer size the more filter rules can be saved in the buffer and thus the higher the hit rate can be maintained.

In the rest of simulation we use buffer size of 16384 byte in the simulated PFC since it performs reasonably well. We study the impact of caching unmatched flows in PFC by making different percentage of unmatched flows go through PFC. Fig. 9 shows the hit rate of PFC under traffic flows with arrival interval uniformly distributed from 5 to 15. Here, all matched means we cache all unmatched flows so that there are no unmatched flows there. As we can see from the Fig. 10, the higher percentage of flows is unmatched, the lower hit rate becomes if unmatched flows are not cached. On the other hand, when the unmatched flows are cached, the hit rate has the highest value.

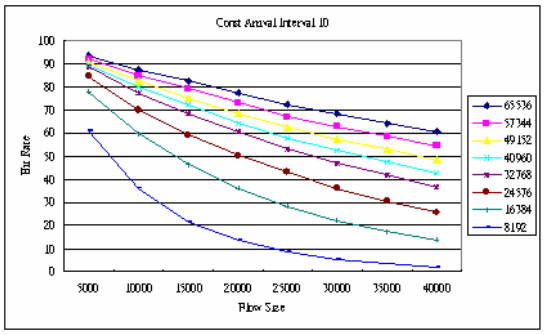

Fig. 8: Different Buffer Size and Flow Size

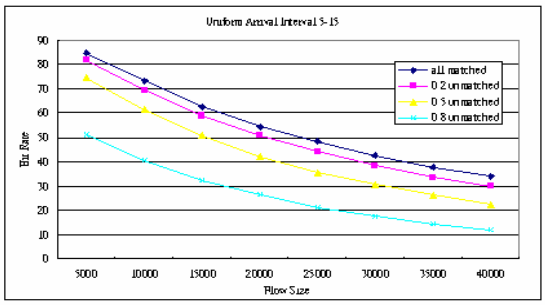

Fig. 9: Different Unmatched Cache Rate with Uniform Distribution

3.4. Different Similar Rate Simulation

When a large percentage of flows match a small set of filter rules, PFC should have very good performance. To study the impact of filter matching percentage, we generate some flows that match the same rules. The traffic arrival interval of these flows is uniformly distributed from 5 to 15. Here, 0.8 similar means 80% of flows hit 10% of packet filter rules. The simulation results shown in Fig. 10 indicate the more flows matching a filter rule, the better is the hit rate of PFC. The results also show even 20 percentage of flows appears a matching pattern, the hit rate improves near 20 percent over when the traffic flows do not show a matching pattern.

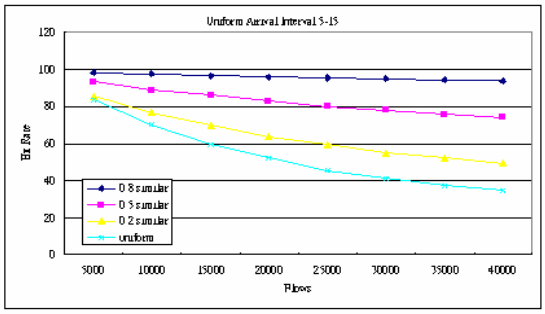

Fig. 10: Different Similar Rate

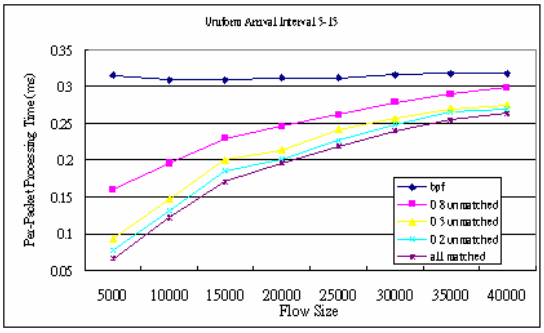

Fig. 11: Per-Packet Processing time in unmatched cache PFC versus BPF

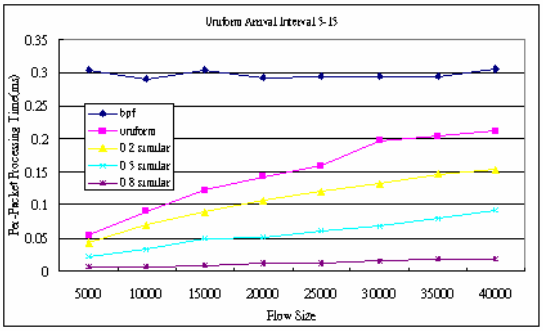

Fig. 12: Per-Packet Processing time in similar flow PFC versus BPF

3.5. Processing Time Simulation

The hit rate of PFC can be translated into processing time overhead. Fig. 11 and 12 shows the processing per packet in BPF and PFC, where in PFC the buffer size is 16384 bytes and the packet arrival interval is uniformly distributed from 5 to 15. As we can see from Fig. 11, at 5000 flows PFC with unmatched caching only uses less than 5 percentage of processing time than that of BPF. As the number of flow increases, higher frequency of hash collision occurs in PFC and thus the average per packet processing also is getting close to that of BPF. Fig. 12 shows the average per packet processing time of BPF and PFC under different degree of filter matching patterns. From the results, we see that PFC needs only half per packet processing time of BPF when 20 percent of traffic flows show a filter matching pattern at 40000 flows. The higher percent of traffic flows show a filter matching pattern, the more significant reduction of packet processing time can be achieved by PFC. Overall, Fig. 11 and 12 shows that PFC performs very well in comparison with tradition BPF without cache.

4. Conclusion and Future Work

In this paper, we propose a new high-performance packet filter architecture named Packet Filter Cache for network monitoring tools. PFC adds a filter rule cache before an existing packet filter. The filter rule cache stores the hash value of a filter rule as a hash table entry that can be searched in O(1) memory access. By taking advantage of the hash lookup speed, PFC can significantly boost filtering performance. The design of hash lookup and checksum verification also can achieve Least Recently Used effect. Moreover, PFC also caches unmatched packet flows to achieve high hit rate. Through analysis and computer simulation we show that PFC can significantly reduce processing time. It improves the processing time about four times at cache hit rate of 70%. Since PFC is an add-on cache architecture, it is very flexible and is readily applied on any existing packet filter without re-engineering existing filter module.

References

[1] J. Mogul, R. Rashid, and M. Accetta, "The Packet Filter: An Efficient Mechanism for User-level Network Code," *In Proc. of the Eleventh ACM Symposium on Operating Systems Principles*, pp. 39-51, November 1987.

[2] EtherReal, http://www.etherreal.org/.

[3] Tcpdump/libpcap, http://www.tcpdump.org/.

[4] N. Brownlee. "Traffic Flow Measurement: Experiences with NeTraMet," *IETF RFC 2123*, March 1997.

[5] L. Deri and S. Suin, "Effective Traffic Measurement Using ntop," *In IEEE Comm. Mag.*, vol. 38, no. 5, pp. 138-145, May 2000.

[6] M. Roesch, "Snort - Leightweight Intrusion Detection for Networks," *In Proc. of the 1999 LISA Conference*, Nov. 1999.

[7] J. Reumann, H. Jamjoom, and K. Shin, "Adaptive Packet Filters," *In Proc. of Global Telecommunications Conference*, pp. 25-29, Nov. 2001.

[8] S. Ioannidis, K. G. Anagnostakis, J. Ioannidis, and A. D. Keromytis, "xPF: Packet Filtering for Low-cost Network Monitoring," *In Proc. of the IEEE HPSR 2002*, pp. 121-126, May 2002.

[9] K. G. Anagnostakis, S. Ioannidis, S. Miltchev, J. Ioannidis, M. Greenwald, and J. M. Smith, "Efficient Packet Monitoring for Network Management," *In Proc. of IFIP and IEEE NOMS 2002*, pp. 423-436, April 2002.

[10] D. R. Engler and M. F. Kaashoek, "DPF: Fast, Flexible Demultiplexing Using Dynamic Code Generation," *In Proc. of ACM SIGCOMM Computer Comm. Review*, vol. 26, no. 4, pp. 53-59, Aug. 1996.

[11] A. Begel, "Applying General Compiler Optimizations to a Packet Filter Generator," *Unpublished Draft*, in http://www.cs.berkeley.edu/~abegel/cs265/cs265-project.ps, 1995.

[12] A. Begel, S. McCanne and S. L. Graham, "BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture," *In Proc. of ACM SIGCOMM Computer Comm. Review*, vol. 29, no. 4, pp. 123-134, Aug. 1999.

[13] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss, "Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages," *In Proc. of the 1994 Winter USENIX Conference*, pp. 153-165, Jan. 1994

[14] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar, "Pathfinder: A Pattern-based Packet Classifier," *In Proc. of USENIX OSDI '94*, pp. 115-123, Nov. 1994.

[15] V, Srinivasan and G. Varghsee, "Fast address lookups using controlled prefix expansion," *In ACM Tran. on Computer System*, pp. 1-40, 1999.