Exploiting Traffic Localities for Efficient Flow State Lookup

Tao Peng, Christopher Leckie, and Kotagiri Ramamohanarao

Department of Computer Science and Software Engineering
The University of Melbourne
Victoria 3010, Australia
{tpeng, caleckie, rao}@cs.mu.oz.au

Abstract. Flow state tables are an essential component for improving the performance of packet classification in network security and traffic management. Generally, a hash table is used to store the state of each flow due to its fast lookup speed. However, hash table collisions can severely reduce the effectiveness of packet classification using a flow state table. In this paper, we propose three schemes to reduce hash collisions by exploiting the locality in traffic. Our experiments show that all our proposed schemes perform better than the standard practice of hashing with overflow chains. More importantly, our *move and insert to front* scheme is insensitive to the hash table size.

1 Introduction

The Internet is playing an increasingly important role in our society. A number of Internet applications, such as proxy services, firewalls and traffic billing need packet classification. In particular, due to the poor security level of the Internet, firewalls are an indispensable component for safeguarding on-line services. Due to the rapid increase in Internet bandwidth, processing speed has become a critical issue for firewall products. Although, firewalls based on FPGA hardware can provide high throughput, they are expensive and inconvenient to upgrade. A more flexible approach is to implement firewalls in software. The benefits of this solution are its low cost and ease of maintenance. However, high packet throughput becomes an issue. Consequently, the challenge is how to maintain high processing speed without compromising the sophistication of the firewall rule set. One important bottleneck in firewalls is caused by rule matching. In this paper, our aim is to exploit locality patterns in Internet traffic to increase the packet processing speed of firewalls.

A common technique to improve firewall performance is to use a flow state table. The firewall maintains a table of all the flows that it has authenticated. Before a packet is sent for rule checking, its flow information is checked to determine whether it appears in the flow state table. If the flow is in the table, the flow has already been seen, and so the packet is handled according to the classification in the table. If not, the new flow is sent for rule checking, and the authenticated flow will be added into the flow state table. Generally, there are two types of data structures that are used to implement the flow state table, namely, hash tables and tree structures. Generally, hash tables are faster than tree structures for insertion and retrieval. A detailed comparison between hash tables and tree structures is outside the scope of this paper. In this paper, we focus on the performance of hash tables for implementing flow state tables.

Our contribution in this paper is to propose three hash table management schemes to improve the processing efficiency of flow state tables by exploiting traffic localities. The rest of the paper is organized as follows. In Section 2, we briefly review the use of hash tables for traffic flows. In Section 3, we propose three hash chain management schemes for handling the hash collisions. In Section 4, we use real-life packet traces to evaluate our schemes. In Section 5, we discuss other related issues for flow state tables.

2 Background on Flow State Tables

Internet services are carried via application sessions, such as downloading a web page or sending an email. Generally, one application session comprises several network sessions, such as TCP sessions or UDP sessions. All the packets involved in one network session belong to one IP flow. In this paper, unless otherwise stated, when we use the term flow we are referring to an IP flow. In IP v.4, we use 13 bytes as the flow identification, which includes the source address, source port number, destination address, destination port number, and protocol number, whereas in IP v.6 [4], we use the flow label and the source and the destination addresses. In this paper, we focus only on IP v.4, but the techniques developed here are also applicable to IP v.6.

We define the number of packets in each flow as the flow length. The flow length varies according to the type of application session. Some flows are short, such as a DNS lookup; some flows are long, such as transferring a large file via FTP. A common operation in network equipment is packet flow classification, for example, for security clearance and traffic management. With the rapid increase in network traffic speed, the computational overhead spent on packet classification has become a hurdle for achieving high packet throughput rates. Fortunately, all packets in the same flow share the same characteristics, such as integrity, priority and routing path. Hence, once one packet from a flow is classified, the rest of the packets within the same flow can share the same classification. This provides us with an opportunity to speed up the packet classification process by focusing only on flow classification.

A flow state table is one type of application that takes advantage of this feature to reduce per packet processing overhead. As shown in Figure 1, each incoming packet is checked to see whether it belongs to any of the flows in the flow state table. If it does, the packet is processed according to the decision associated with the matched flow. If it does not, the packet is classified by checking the whole rule set, and the classification result is used to update the flow state table. As the overhead for flow state table lookup is much smaller than checking the whole rule set [5], the overall packet classification overhead is reduced. Gupta et al. gave an excellent tutorial on packet classification algorithms in [5]. However, to our best knowledge, no research has been done at the time of writing this paper on reducing the cost of flow state table lookup.

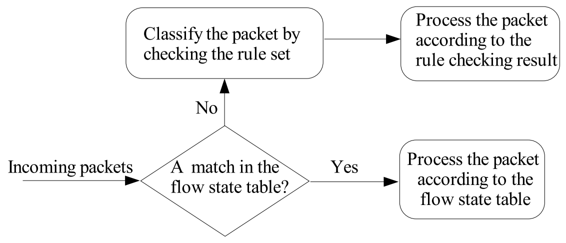
Classify the packet by checking the rule set
Process the packet according to the rule checking result
No
Incoming packets
A match in the flow state table?
Yes
Process the packet according to the flow state table

Fig. 1. Flow state table and packet classification

Several data structures have been proposed for flow state table lookup, e.g., search trees and hash tables [6] [3]. Typically, hash tables have the advantage of fast lookup (O(1) if no hash collision happens), while search trees have the advantage of dynamically adjusting the size of the data structure to the amount of data that needs to be stored.

Our focus is on using a hash table to store the flow state information. Typically, the 13-byte flow information f is first reduced using a hash function $h(f)$, which returns an index into the hash table. If the hash table has N entries, then the hash function returns an index in the range $0 \leq h(f) < N$. In practice, multiple flows can have the same hash index. Consequently, each entry in the hash table, known as a *bucket*, is a pointer to a linked list of flow records that have all been hashed to that entry. When two different flows map into the same bucket, a *collision* has occurred. These linked lists are also referred as *hash chains.* A critical issue for the performance of insertion and retrieval is how to manage records in the linked list.

For example, one conventional approach taken by a well-known open source firewall package IPfilter [6] is to always insert the new flow state record at the front of the linked list. The idea behind this approach is that once the new flow state is inserted, it is likely to be followed by packets from the same flow. Hence, most of the packets will match the first node of the hash chain. Unfortunately, tens of thousands of flows can be active at the same time in practice. In particular, several flows that are mapped into the same hash bucket can be active at the same time. Generally, these flows interleave with each other. For example, let A_i represent one packet from flow A, B_i represent one packet from flow B, C_i represent one packet from flow C, and flows A, B, C are mapped into the same hash bucket. For IP filter, the ideal sequence of the three incoming flows will be $A_1A_2A_3A_4A_5...B_1B_2B_3B_4B_5...C_1C_2C_3C_4C_5....$ In practice, the sequence is more likely to be interleaved, e.g., $A_1B_1A_2A_3A_4B_2B_3B_4C_1C_2A_5B_5C_3C_4C_5....$ In this scenario, the traffic locality cannot be fully exploited by just inserting the new node at the front of the list.

Let us consider another extreme example. Assume four active flows are mapped into the same hash bucket. All these four flows are active for the same time period. Three of them are slow flows, e.g., ssh sessions, and one of them is a fast flow, e.g., a large file download. We assume the fast flow starts *before* the three slow flows. Once these four flows are included in the *hash chain*, their positions in the hash chain will never be changed given no new flows are mapped into the same hash bucket. As shown in Figure 2, the fast flow who has the dominant number of packets of these four flows will have to search to the end of the list to find a matching flow. In this scenario, most of the packets will experience the longest searching time. Consequently, due to the complicated packet arrival sequence, simply inserting the new flow state at the front of the list is not enough to guarantee fast lookup performance. Better algorithms are needed to carefully exploit locality in the traffic flows.

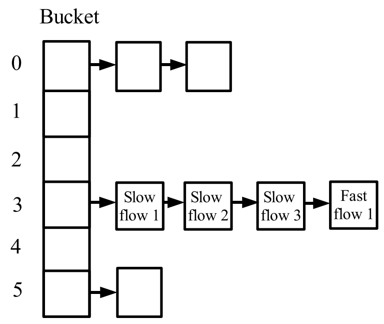

Fig. 2. Worst-case scenario for the conventional approach to managing the hash chain.

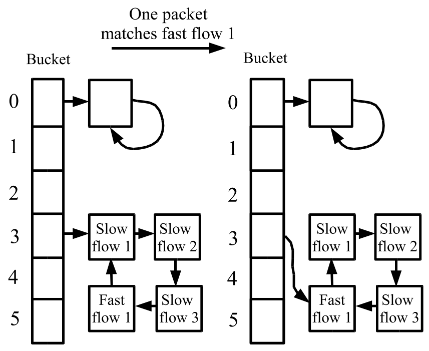

Fig. 3. The circular list hash chain

3 Proposed Schemes for Flow State Table Management

Our aim is to reduce the time needed for flow state table lookup, i.e., hash table lookup, so that the overall packet processing speed is increased. The key step to improving the hash table lookup speed is to organize the hash chain so that most of the packets only need a few searches to find a matching flow. As we discussed in Section 2, we need a more adaptive scheme to exploit traffic locality. Our assumption is that for each hash bucket, the packets from the same flow will arrive continuously for at least two packets. We call these continuously arriving packets from the same flow a *burst*. For example, the sequence $A_1B_1A_2A_3A_4B_2B_3B_4C_1C_2A_5B_5C_3C_4C_5$... contains several bursts, such as $A_2A_3A_4$ and $B_2B_3B_4$. The conventional approach only focuses on organizing the hash chain according to the first packet of each flow, and fails to exploit the locality within each burst. To address this shortcoming and inspired by Zobel's idea in accumulating text vocabularies [7], we propose three different algorithms focusing on exploiting the locality within a burst.

– *Move to front*: This scheme maintains a linear list, and moves the most recently matched node to the front of the list, and inserts new flows at the end of the list.
– *Insert and move to front*: This scheme also maintains a linear list. However, it not only moves the matched flow to the front but also inserts the new flow to the front.
– *Circular list*: This scheme maintains a circular list for each hash bucket, and the hash bucket either points to the last matched flow or the newly inserted flow as shown in Figure 3.

All of these three schemes share the same feature that each matched flow will trigger a restructure of the hash chain, i.e., moving the matched flow to the beginning of the list. Hence, at least the second packet of each burst will only need one search operation to find a matched flow. In the scenario shown in Figure 2, once one packet from the fast flow 1 searches till the end of the hash chain to find a match, the fast flow 1 will be moved to the front. Hence, the following packets from the fast flow 1 only need one search to find the matching flow. For the simplicity of comparison, we refer to the approach taken in IP filter as the *normal* scheme, which is described in detail in Section 2 as the conventional approach. In the next section, we will use real-life data traces to verify our heuristics.

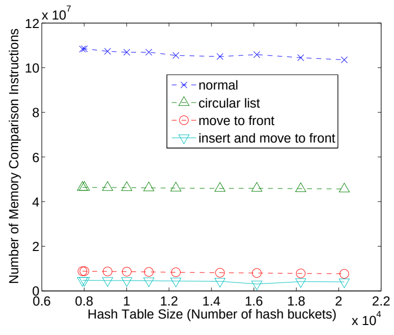

Fig. 4. Number of memory comparison instructions used by each scheme for IP traffic in the Auckland trace.

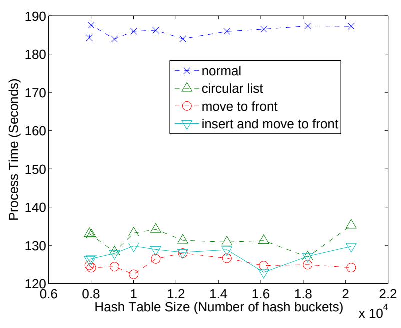

Fig. 5. The process time used by each scheme for IP traffic in the Auckland trace.

4 Evaluation

Our aim is to use real-life packet traces to validate our proposed schemes. There are two sets of packet traces that we use in the evaluation. The first packet trace is collected at the edge router of the Department of Computer Science and Software Engineering of the University of Melbourne in Australia. We refer to these as the *Melbourne traces*. The advantage for this packet trace is that it is recent and the whole IP packet header information is preserved. Unfortunately, these packet traces are not publicly available yet due to the University privacy policy. The second packet trace is collected at the Internet uplink of the University of Auckland in New Zealand [1]. We call these the *Auckland traces.* The Auckland traces are publicly available so that our results on this data trace can be easily reproduced by other researchers. The details of these two data sets are shown in Table 1.

	Melbourne traces	Auckland traces
Direction	bi-directional	uni-directional
Date of Collection	22 May 2004	29 March 2001
Duration	24 hours	24 hours
Total Number of Packets	153,399,408	43,226,495
Format	tcpdump format	DAG format

Table 1. Data sets used in evaluation

Our experiments include on-line evaluation and off-line evaluation. For the Melbourne trace, we conducted an on-line evaluation. In the flow state lookup engine we implemented the normal scheme and our three proposed flow state table lookup schemes. In the traffic generator we use tcpreplay [2], an open source tool, to replay the traffic traces collected. Both the flow state lookup engine and the traffic generator comprise a 2.8 GHz Pentium 4 processor with 1 MB L2 cache and 512 MB RAM. For the Auckland trace, we conducted an off-line evaluation [1] in a linux PC with dual 900MHz Xeon CPUs (each CPU has 512 KB L2 cache), and 512 MB RAM. Instead of reading the traffic from the network card, our programs read the traffic from the data trace file. As the total number of flows in each data set is large and we only need to maintain the states for the active flows, we need a mechanism to delete the old flow states and add the new flow states. For the convenience of evaluation of each hash chain management scheme, we simply remove the last 5 flows states once the hash bucket size reaches 10. The key goal of our experiments is to investigate how our algorithm improves the flow state lookup efficiency. The hash function we use is the bit-wise hash function from Zobel et al. [7].

4.1 Process Efficiency

We use two different metrics to evaluate the efficiency of each flow state lookup scheme. First, we count the number of *memory comparison instructions* each scheme has used. Note that for simplicity, we count the number of memory comparison instructions rather than the total number of instructions executed. In our experiment, we count the number of calls to *memcmp()* used by each scheme. A smaller number of instructions indicates that the traffic localities have

[1] The data server that was used to store the Auckland traces did not have enough resources to conduct an on-line evaluation.

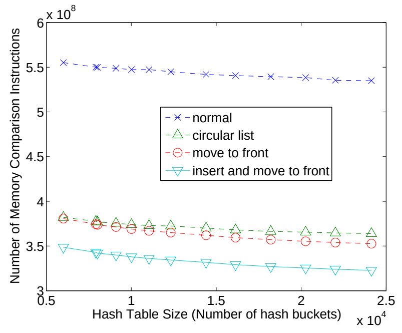

Fig. 6. Number of memory comparison instructions used by each scheme for IP traffic in the Melbourne trace

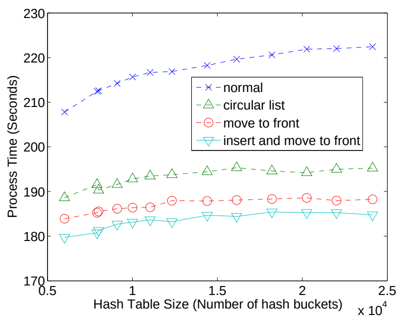

Fig. 7. Processor time used by each scheme for IP traffic in the Melbourne traffic

been better exploited, and the scheme has high efficiency. Second, we record the time spent by the processor (not the elapsed time) to process the same amount of traffic for each scheme. Each scheme only processes the IP traffic. Moreover, we vary the hash table size to see how this affects the performance of each scheme.

Figure 4 illustrates the performance of our schemes for the Auckland traces. The *normal* scheme used 1.1×10^8 memory comparison instructions, the *circular list* scheme used 4.6×10^7 memory comparison instructions, while the *move to front* scheme and the *insert and move to front* scheme used only 8.4×10^6 and 4.5×10^6 memory comparison instructions respectively. Similarly, Figure 6 illustrates the performance of our schemes for the Melbourne traces. The evaluation results are quite similar to the Auckland traces. The *insert and move to front* scheme still performs the best, followed by the *move to front* scheme and *circular list* scheme.

The *move and insert to front* scheme performs the best because it keeps the more recent flow closer to the front of the hash chain. As the *move to front* scheme inserts the new node at the end of the hash chain, the entire hash chain has to be searched to find a matched node when the next packet of the new flow arrives. Hence, the *move to front* scheme needs more memory comparison instructions than the *insert and move to front* scheme. As the *circular list* leaves the pointer at the most recently matched node, the node in front can become the last node. As shown in Figure 3, if an incoming packet matches the slow flow 2, the hash bucket leaves the pointer at the slow flow 2. All packets from slow flow 2 only need one search to find a matching flow. However, if the next incoming packet is from slow flow 1, then it needs 4 searches to find the matching flow entry. This problem can be exacerbated if we have multiple fast flows in the same bucket. Consequently, the number of memory comparison instructions needed by the *circular list* scheme is larger than both the *move to front* scheme and the *insert and move to front* scheme. Figure 5 illustrates the performance of all four schemes in terms of processing time. We can see that the *move to front*

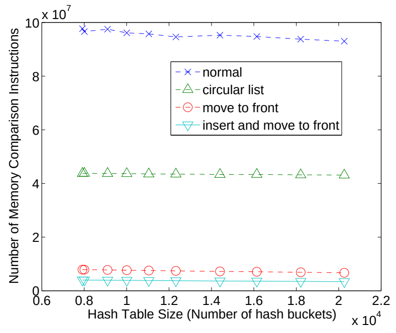

Fig. 8. Number of memory comparison instructions used by each scheme for TCP traffic in the Auckland traces

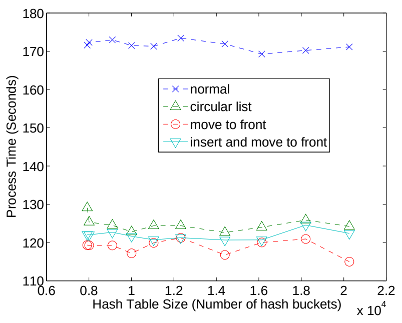

Fig. 9. Processor time used by each scheme for TCP traffic in the Auckland traces

scheme, the *insert and move to front* scheme, and the *circular list* scheme have very similar performance, which is about 30% faster than the normal scheme. The benefit for the *circular list* scheme is that only one operation is needed to let the hash bucket point to the recently matched node, which is simpler than the linear list. This benefit is diminished by the disadvantage of a large number of searches. Consequently, the overall processing time for the *circular list* scheme is very close to the *move to front* scheme and the *insert and move to front* scheme. Similar results are shown in Figure 7. We can see that in the Melbourne trace the *insert and move to front* scheme clearly stands out as the best scheme. More importantly, for hardware implementation, the hash chain can be implemented as a ring buffer instead of a linked list, where the size of the bucket is fixed. In this scenario, the *circular list* is more efficient than the other three schemes as it only needs to keep track of the most recently matched flow, and does not need to keep the rest of the flow entries in order.

4.2 Traffic Type

In some flow state table lookup implementations, e.g. a firewall application, the TCP, UDP, and ICMP flows are treated differently. For example, firewalls will have different sets of rules for each protocol. More importantly, only TCP traffic requires an initial handshake to establish a connection, and is regarded as stateful traffic. Hence, maintaining a flow state table for TCP traffic is essential for many Internet applications. This section investigates how our schemes perform under TCP traffic. Figures 8 and 9 illustrate the results for the Auckland traces. Figures 10 and 11 illustrate the results for the Melbourne traces. We can see that all of our three schemes perform better than the normal scheme. Overall, the *insert and move to front* scheme performs the best.

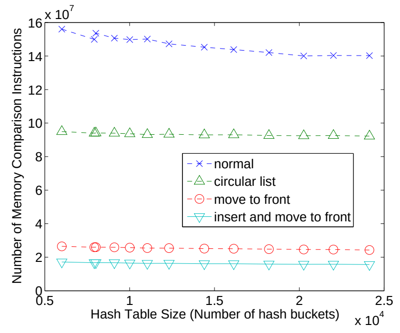

Fig. 10. Number of memory comparison instructions used by each scheme for TCP traffic in the Melbourne trace

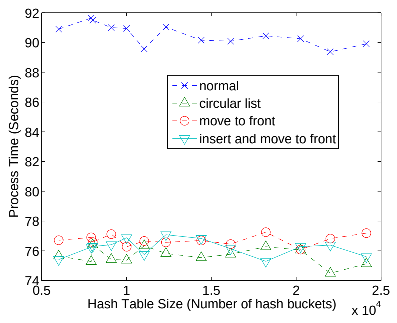

Fig. 11. Processor time used by each scheme for TCP traffic in the Melbourne trace

4.3 Hash Table Size

In this section, we aim to investigate the effects of the hash table size on the performance of each scheme. As we see from the Figures 4 to 11, the flow state table lookup performance is not sensitive to the hash table size. In theory, a larger hash table size can reduce the hash collision rate and the average hash chain length, and hence enhance the flow state table lookup performance. However, in our experiments, the hash chain length is reduced to be 5 once it reaches 10 in order to delete old flows. As the total number of flows is much larger than the number of flows the hash table can hold, each hash chain length is likely to reach 10 and will be reduced to 5 afterward. Hence, the average hash chain length is always between 5 and 10 for hash tables with different sizes. The benefit of a large hash table is to hold more flow states and reduce the number of flows that need to be re-inserted after deletion. The major cost of inserting a new flow is to classify the flow, i.e., deciding whether to admit or reject the flow. This cost is highly application dependent. Consequently, we do not consider this cost in our experiments, and a flow is inserted directly once no matching flow is found. For this reason, the hash table size has little effect on the flow state table lookup performance.

To facilitate our study of the impact of the hash table size, we performed an extreme experiment, where no flows are purged from the hash bucket. However, this experiment is limited by hardware, i.e., the RAM size. We conducted this experiment on TCP traffic in the Melbourne trace, as it contains 2 million flows which can be handled by our hardware. As we see from Figure 12, all our schemes perform significantly better than the normal scheme when the hash table size is small. Generally, the small hash table size will increase the length of the hash chain, which in turn increases the processing time for each flow state lookup. However, this processing time is reduced if we keep reordering the hash chain to make sure all the active flows are in the front. More importantly, the *insert* *and move to front* scheme is insensitive to the hash table size as it fully exploits the temporal localities in the traffic. The processing time of all the four schemes converges when a large hash table size is used. This is due to the fact that the average size of the hash chain is reduced to be 1, and there is no difference between the four schemes.

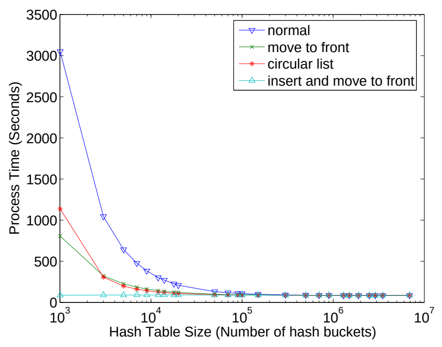

Fig. 12. Processor time used by each scheme for TCP traffic in the Melbourne trace, where no flows are purged from the hash bucket.

4.4 Complexity of the Schemes

It is difficult to discuss the complexities of the schemes without knowing the traffic flow statistics and hash function properties. Let us define a simple uniform hash function[2]

$$h : K \longmapsto T$$

where K is the set of keys (the 13 bytes flow identification in our experiment), where T is the set of memory locations in the hash table. Let n be the hash table size and l be the number of keys we want to hash, then the *load factor* can be defined as $\alpha = \frac{l}{n}$. Based on our observation on the Auckland traces and Melbourne traces, we assume the packet sequence within each hash bucket is as follows:
$A_1A_2...A_kB_1B_2...B_kC_1C_2...C_k...A_{(j-1)k+1}A_{(j-1)k+2}...A_{jk}B_{(j-1)k+1}B_{(j-1)k+2}...$ $B_{jk}C_{(j-1)k+1}C_{(j-1)k+2}...C_{jk}(j \gg 1, k \gg 1)$, where A, B, C are the flows that are mapped into the same hash bucket, k is the size of the *burst* and j is the number of bursts in each flow. Then we can summarize the searching complexity of the normal scheme and our schemes in Table 2, where X_i $(i = 0, 1, 2, 3)$ is a variable representing the cost of searching for an element in a hash chain, and Y_i $(i = 1, 2, 3)$ is a variable representing the cost of reordering the hash chain. Note that $i = 0$ is the *normal* scheme, $i = 1$ is the *move to front* scheme, $i = 2$ is the *circular list* scheme and $i = 3$ is the *insert and move to front* scheme. Based

[2] A *Simple Uniform Hash function* assumes that any given element is equally likely to hash into any one of the hash buckets.

on the analysis of Section 4.1, the variables satisfy $X_0 \approx X_2 > X_1 > X_3$ and $Y_1 = Y_3 > Y_2$. We can see that all our schemes outperform the normal scheme in terms of complexity if the burst length k is sufficiently large.

Normal	Move to front	Circular list	Insert & move to front
$O(1 + X_0\alpha)$	$O(1 + \frac{1}{k}X_1\alpha + \frac{1}{k}Y_1)$	$O(1 + \frac{1}{k}X_2\alpha + \frac{1}{k}Y_2)$	$O(1 + \frac{1}{k}X_3\alpha + \frac{1}{k}Y_3)$

Table 2. Comparison between our three schemes and the normal scheme in terms of searching complexity.

5 Discussion

From Figure 8 and Figure 10, we can see that our three proposed hash chain management schemes perform better in the Auckland traces than in the Melbourne traces. For example, in the Auckland trace, the circular list scheme reduced memory comparison instructions by 55% compared with the normal scheme; while in the Melbourne trace, it only reduced the memory comparison instructions by 40% compared with the normal scheme.

The major reason for this performance difference is the difference in traffic characteristics. As we analyzed in Section 3, our three proposed schemes perform well if the flows are not highly interleaved, i.e., if the packets of each flow tend to arrive in uninterrupted bursts. The Auckland trace is uni-directional while the Melbourne trace is bi-directional. For the TCP traffic in the bi-directional trace, the incoming flows and the outgoing flows are highly correlated. Hence, the flows in the Melbourne trace are more heavily interleaved than the Auckland trace.

More importantly, the Melbourne trace is collected at the router of the computer science department in the University of Melbourne. Hence, it contains a lot of inter-departmental traffic, and only 50% of the total traffic is TCP traffic. Moreover, nearly 16% of the TCP traffic is SSH traffic. In contrast, the Auckland traces are collected at the router that provides the Internet connection to the University of Auckland. More than 95% of the Auckland trace is TCP traffic, while only 0.25% of the TCP traffic is SSH traffic. Generally, SSH traffic is generated by users who remotely login to a server. The speed of the SSH flows are strongly affected by many human factors, e.g., a person's typing speed or the time spent thinking between typing commands. Moreover, the SSH flows can be active for quite a long time, e.g., researchers may login to servers for several days to run some time-consuming experiments. In summary, the SSH traffic can be described as slow speed flows that last for a long time. Consequently, we are less likely to see flow bursts in the SSH traffic, as slow flows are more likely to be highly interleaved.

From Figure 4 and 8, we can see that the number of memory comparison instructions used in TCP traffic is close to the number of memory comparison instructions used in IP traffic. This is because the majority (over 95%) of traffic in the Auckland traces is TCP traffic. From Figures 6 and 10, we can see that the number of memory comparison instructions used in TCP traffic is less than 20% of the number of memory comparison instructions used in IP traffic. This can be explained by the following two reasons. First, the TCP traffic only accounts for 50% of the total IP traffic in the Melbourne traces. Second, we found a large proportion of UDP scan traffic in the Melbourne traces. The UDP scan traffic generally consists of a large number of short UDP flows, which adds extra overhead to the flow state lookup for the Melbourne traces.

6 Conclusion

In this paper, we have proposed three hash chain management schemes to improve flow state table lookup performance by exploiting the temporal locality in network traffic. Our *insert and move to front* scheme which inserts the new flow and moves the matched flow to the front of the hash chain, performs the best. It is followed by our *circular list* scheme and *move to front* scheme. The *insert and move to front* scheme has the best performance and is insensitive to the hash table size as it constantly reorders the hash chain to accurately reflect the locality of the incoming traffic. The circular list approach only preserves the locality for the matched node, where the localities of rest of the flows in the bucket are lost. By evaluating our schemes using real life data traces, we see that all our three schemes perform better than the normal scheme. We also discussed several factors that affect the flow state table lookup performance, such as the type of traffic. The conclusions of our research are now being incorporated by our industry partner (Intelliguard Pty Ltd) into their network security products.

Acknowledgment

We would like to thank the ARC Special Research Center for Ultra-Broadband Information Networks (CUBIN) for providing the test environment, and thank the Waikato Applied Network Dynamics Research Group for making available their data traces. We also wish to thank Mr. Darren Reid and Mr. Peter Yandell for their useful suggestions and comments. This work was supported by the Australia Research Council.

References

1. Waikato Applied Network Dynamics Research Group, The University of Waikato.
2. http://tcpreplay.sourceforge.net/.
3. Packet Filter. http://www.benzedrine.cx/pf.html.
4. S. Deering and R. Hinden. Internet protocol, version 6 (IPv6) specification. RFC 2401, the Internet Engineering Task Force (IETF), December 1998.
5. P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network*, March 2001.
6. Darren Reed. IP Packet Filter. http://coombs.anu.edu.au/~avalon/.
7. J. Zobel, S. Heinz, and H. E. Williams. In-memory hash tables for accumulating text vocabularies. *Information Processing Letters*, 80(6):271–277, 2001.